Personal Computer

NEC PC 8801

アセンブリ言語
プログラミング入門

脇英世＝著

ナツメ社

[本書を読まれる方へ]

⊙アセンブラはむずかしい、と、よくいわれます。
確かにそういった一面もありますが、
それは初めのうちだけです。
いいかえれば、なれが必要ということなのです。
⊙アセンブリ言語プログラミングを勉強すると、
どんなメリットがあるのでしょうか。
事務処理の合理化が要求されるビジネスの分野では、
ベーシックと比べて、ずっとスピード・アップがはかれます。
とくにデータの並べかえなどでは、
その差が顕著にあらわれます。ベーシックの
1000倍くらいの速さでソートできるのですから。
⊙さいわいPC-8801には、
この便利なアセンブラ機能が内蔵されています。
ちょっとした努力で理解できるのですから、
せっかくあるものを利用しない手はありません。
⊙とくにこの本は、
アセンブラ知識の用意がない人を対象にしています。
とにかく、手がけてみることです。
そしてそれぞれのステップを、
書いてあるとおりに打ち込んでみてください。
知らず識らずのうちに興味がわいてくるはずです。

cover design——nobuo nakagaki + yoshifumi hayase

Personal Computer

PC NEC 8801

アセンブリ言語
プログラミング入門

脇英世＝著

ナツメ社

はじめに

１９８２年の３月ナツメ社の社長の田村さんと編集部の黒田さんがお見えになりました。ちょうど卒業式の日で大変あわただしく、ゆっくりお話もできませんでしたが、アセンブリ言語の本を書くようにとの御依頼でした。ナツメ社のもともとの御注文はＰＣ-６００１用のアセンブリ言語の本でしたが、私の希望でＰＣ-８８０１用になりました。

ところが私が日本語ワープロの仕事で忙しく、なかなか時間がとれず本格的に仕事に入れたのは９月に入ってからです。しかし筆はなかなか進まず、黒田さんの熱心な督促に閉口したこともあります。私がやる気を出したのは社長の田村さんがパソコンに興味を持っていて、足しげく私のところへおいでになり頻繁に接触するようになってからです。田村さんの質問は大変に鋭く良くポイントをついていて、勉強熱心なのには頭が下がりました。これは大変はげみとなり、当初の悲観的予想よりもずっと早く脱稿できることになりました。

この本の考え方は特に難しい技巧を教えることではなく、やさしい例題をつみ重ねていくことと再現性の良さを考慮したことです。この本に書いてある通りに入力すれば必ず動作するようにＰＣ-８８０１のモニタプログラムを利用し、特に高価なアセンブラプログラムを買わなくともすむようにしました。

書き上げてしまいますと、もっと書き込んでおけば良かったとか、あそこはこうすべきであったとかいろいろ思い悩みますが、今後改訂の機会があれば徐々に改良していきたいと思います。

１９８２年１２月２４日

脇　英世

もくじ

第0章

アセンブリ言語プログラミング

0・1　難しそうなアセンブリ言語

アセンブリ言語とは何でしょうか？ＢＡＳＩＣが相当できるようになった人でもアセンブリ言語と聞くと、いやだなあと思うのではないかと思います。実際アセンブリ言語は難しそうに見えます。たくさんの記号の列や、数字の列がずらりとならんだプログラムを見ると、だれでもゾッとするのではないでしょうか？

ここで用語について一言申し上げておきますと、アセンブリ言語というとき、いくらか混乱があるようです。

みなさんもよくご承知のように、コンピュータは０と１の電気信号しか理解できません。従って、コンピュータにかわる形でプログラムを書くには、０と１だけしか使ってはいけないのです。このため０と１だけで書かれたプログラムが存在します。これを**機械語プログラム**といいます。

機械語プログラム

●０と１だけで書かれたプログラム

ところが、０と１だけで書かれた機械語プログラムは、人間にとってはたいへんわかりにくいものです。次をごらんください。

```
AF 32 10 E0 76
```

このため、人間にとってもうすこしわかりやすいかたちのプログラムが要求されます。これが**アセンブリ言語**で書かれたプログラムです。これは機械語プログラムに比べて、少しわかりやすくなっています。次をごらんください。

アセンブリ言語のプログラム

●人間にわかりやすいような記憶用のコードで書かれたプログラム

```
XRA A
STA E010
HLT
```

このアセンブリ言語のプログラムから、機械語プログラムに変換するプログラムもあります。本当はこのプログラムのことを**アセンブラプログラム**というのです。ところが実際には、アセンブリ言語のプログラムも、また場合によっては機械語プログラムでさえもアセンブラプログラムとよばれることもあります。正式には３つを厳密に区別しなければなりませんが、ここではあまりやかましいことをいわないことにしましょう。

アセンブラプログラム

●アセンブリ言語のプログラムから機械語のプログラムへ翻訳するプログラム

0・2　なぜ必要か？

アセンブリ言語はやさしいという人もいますが、正直なところをいえば、そんなにやさしくありません。ＢＡＳＩＣのような高級言語で書かれたプログラムに比べて、アセンブリ言語で書かれたプログラムは難しいことも事実です。

なぜアセンブリ言語は難しいのでしょうか？　それはＢＡＳＩＣのような高級言語で書かれたプログラムの場合は、マイクロコンピュータの本体であるマイクロプロセッサの構造に関する知識などが全く必要なかったのに比べ、アセンブリ言語の場合はある程度マイクロプロセッサの内部構造に関する知識が必要になったりするからです。たとえば、これから勉強していくＰＣ－８８０１の場合、ＣＰＵ（中央情報処理装置）であるマイクロプロセッサＺ８０のレジスタがいくつあるかとか、インデックスレジスタがいくつあるかとか、ハードウェアの知識も必要になることもあるからです。

つぎに、実際にやりたいことと、アセンブリ言語で書かれる内容には大きなへだたりがあることがあげられます。

たとえば、みなさんがゲームのプログラムを作りたいと考えているのに、アセンブリ言語ではレジスタ間のデータの転送や、メモリへのデータの書き込みや、読み出しといったことしかできません。

アセンブリ言語プログラムを見てみますと、これがどうしていろいろな絵を動かしたり、さまざまな音を出すのか不思議に思われるでしょう。

では、なぜアセンブリ言語プログラムを勉強しなければならないのでしょうか？　これに対するはっきりした答を与えるのは難しいですが、ＢＡＳＩＣで書かれたプログラムは、実行速度がたいへん遅く、またたくさんのメモリを必要とします。これに比べてアセンブリ言語プログラムは非常に高速で、ゲームのように瞬間的な応答を必要とする場合や、微妙なコントロールを必要とする場面にはなくてはならないものですし、メモリも大幅に節約することができます。すべてのプログラムをアセンブリ言語で書く必要はありませんが、ＢＡＳＩＣと組み合わせることにより非常に豊かな可能性が生まれてきます。

高級言語

●0と1だけで書かれる機械語に比較して人間にとってわかりやすい言語。コンピュータにとっては高級だが人間にとってはやさしい

マイクロプロセッサ

●超小型の情報処理装置のこと。マイクロ・コンピュータの本体をなしている。40本足のゲジゲジ虫の形をしている。

CPU

(Central Processing Unit)
●中央情報処理装置とこと。コンピュータの核心をなす。

0·3 BASICとアセンブリ言語

ＢＡＳＩＣで書かれたプログラムは、多少手直しをすれば他のパーソナルコンピュータでも動かすことができます。

たとえば、ＰＣ-８８０１用に書かれたＢＡＳＩＣプログラムは、ある程度の手直しをすることによってＦＭ８用にも書き直すことができます。また、アップル用にも書き直すことができます。

ところがＰＣ-８８０１用に書かれたアセンブリ言語プログラムは、ＦＭ８には移植できませんし、アップル用にも移植できません。

プログラムの流通性ということから考えますと、非常に不便なように思えますが、それだけマイクロプロセッサの特性を十分に利用していることになるので、より高度なプログラムを少ないメモリで動かすことができるのです。

とはいっても、画面に直線や円を書くプログラムや、文字列の処理のプログラムを自分で作ることは勉強にはなりますが、あまり感心しません。手間ばかりかかって間違いだらけのプログラムができることうけあいです。

つまり、アセンブリ言語プログラムは決定的な局面において使う秘密兵器のようなものです。プロの人はすべてアセンブリ言語で書くのがよいと思いますが、アマチュアの場合には、ＢＡＳＩＣのよいところとアセンブリ言語のよいところを上手に組み合わせて使うほうがよいと思います。

アセンブリ言語プログラムを勉強することは、あなたのコンピュータの世界をずっと広げることになります。

ＢＡＳＩＣにご注意

ＰＣ-８８０１のＢＡＳＩＣはみなさんもよくご存知のように、マイクロ・ソフト社のウイリアム・Ｈ・ゲーツという天才によって書かれました。ゆきとどいた配慮とＢＡＳＩＣの高速性で全世界を屈服させたのですが、科学技術計算に関してはいくらか甘い点があるといわれています。

たとえば、計算の精度でも単精度６ケタぐらいですが、このままくり返しの数値計算をするのは危険です。累積誤差が大きくなります。必ず倍精度を使わねばなりません。また、組み込まれている三角関数も比較的簡単な近似式を使っており、場合によっては大きな誤差がでるようです。従って、本当に厳密な計算をする場合には十分注意を払わねばなりません。

0·4　予備知識はいりません

おどかすようなことばかりをいいましたが、これからアセンブリ言語プログラムの勉強するにあたっては何の予備知識も必要としません。

やさしいところからむずかしいところへ徐々にみなさんをご案内していくつもりです。

ていねいに説明していきますから、みなさんも根気よく読んでください。そして、できればＰＣ-８８０１に実際に入力してみてください。この本のプログラムはすべてＰＣ-８８０１で実行したものばかりですから、全く同じにキーをたたいていけば、理屈はわからなくても動かすことはできます。

最初から何もかもわかろうとする必要はありません。はじめは真似から出発して少しずつ変形し、新しい試みをしてください。いろいろ失敗も起きると思いますが、決して失望したり、落胆しないでください。こういう種類のミスはどうして起きるかということをたくさん知っているほど、プログラミングは上達するのです。

ＢＡＳＩＣと機械語のスピードの違い

機械語を使ってプログラムを作ると、ＢＡＳＩＣに比べてどのくらいスピードが向上するでしょうか。プログラムを作る人の腕にもよりますが、機械語の場合、１つの命令の処理には１μＳ（マイクロセカンド）かかり、ＢＡＳＩＣの場合１つの命令の処理に１ｍＳ（ミリセカンド）かかると大ざっぱな見積りを持っておくとよいでしょう。

ＭＳは10^{-6}秒ですから、１秒間に10^{6}個＝百万個の命令を処理しますが、ｍＳは10^{-3}秒ですから、１秒間に10^{3}個＝千個の命令が処理されると考えてよいでしょう。理屈では機械語はＢＡＳＩＣの千倍のスピードで走るのですが、プログラムを作るのはなかなか大変です。千倍のスピードを得るには千倍とまではいかなくとも百倍ほどの人間の努力が必要となります。世の中はうまくしたもので、どこかで努力が必要となるのです。

0・5　準備するもの

これから勉強を始めるにあたって、つぎのものをみなさんが持っているものと仮定して話をはじめます。

まず、ＰＣ-８８０１本体、これはいうまでもなく絶対に必要です。

つぎにＣＲＴディスプレイですが、高解像度のカラーＣＲＴや標準カラーＣＲＴは必要ではありません。(持っていたほうが楽しく便利だと思いますが、アセンブリ言語プログラムの勉強には当面必要ではありません。)

フロピーディスクはあったほうがよく、８インチのフロッピーディスクは非常に使いやすく便利ですけれども、なくてもかまいません。

ミニフロッピーもなくてもかまいません。通常のカセットテープレコーダで結構です。アセンブリ言語を本当に使いこなすには、８インチのフロッピーディスク装置があってＣＰ／Ｍというオペレーティングシステムが使えるとたいへん便利なのですが、こうした高級なプログラムや機器を使いこなすには、まず本書でアセンブリ言語プログラムの基礎を勉強されてからのほうがよいと思います。また、市販されている本式のアセンブラプログラム、たとえばマイクロソフト社のアセンブラプログラムを使いこなすことは、本書の目標には入っていません。

ＣＰ／Ｍ
オペレーティングシステム

つぎに、プリンタはぜひ１台ほしいと思います。高級なものでなくて結構ですが、プリンタがありませんとプログラムの虫（バグ）を発見するのに不便です。もちろんＣＲＴディスプレイ上で表示させるだけでも相当のことはできますが、こうしたソフトコピーというやり方は一画面分しか見ることができず、記録性もないので、ぜひプリンタは１台そなえてください。

バグ
●南京虫のことで、プログラムのあやまりをいう

コーディングシート

最後に、コーディングシートと呼ばれるプログラム設計用のシートを準備してください。大きさのきまった紙にキチンとコーディングし、保存しておくことが大切なことです。ともすれば、ありあわせの紙の裏にクシャクシャと走り書きする人を見かけますが、決して感心した習慣ではありません。お金を出して買わなくても自分で作ればよいでしょう。次ページにコーディングシートの一例を示しますから、みなさんも自分で工夫して作ってみてください。コピーして何十枚か持っていると大変便利です。

第1章

PC-8801の基礎知識

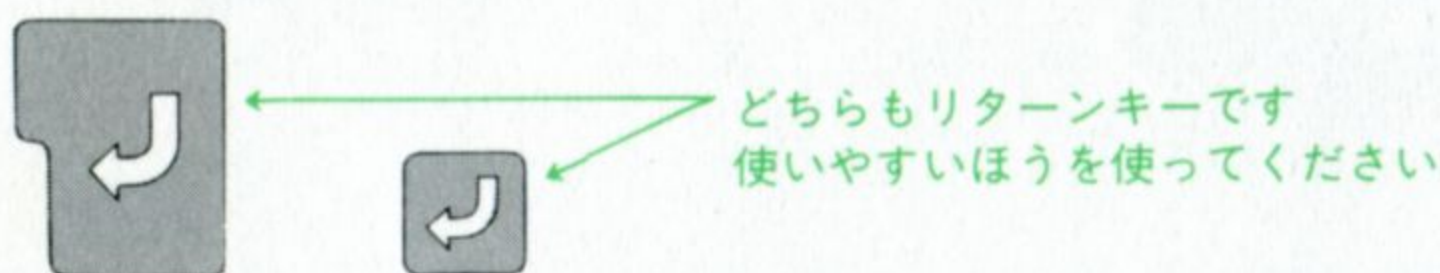
◆

1・1　キーボードの使い方

アセンブリ言語プログラミングを勉強しようというほどのみなさんに、キーボードの使い方の説明は不用かもしれませんが、他のパソコンを使っていてＰＣ－８８０１にうつられた方にとっては目新しいこともあるかもしれませんので、いくつかご注意しておきましょう。

リターンキー（⏎）

まず次ページにＰＣ－８８０１のキー配列を示しておきます。大体は通常のパソコンと同じです。ＰＣ－８００１と比較すると、ＲＥＴＵＲＮ（リターン）キーが左下を向いた曲がった矢印になっていることが目をひきます。矢印になっていても、ＲＥＴＵＲＮの機能であることには違いがありません。

どちらもリターンキーです
使いやすいほうを使ってください

コントロールキー
（CTRL）

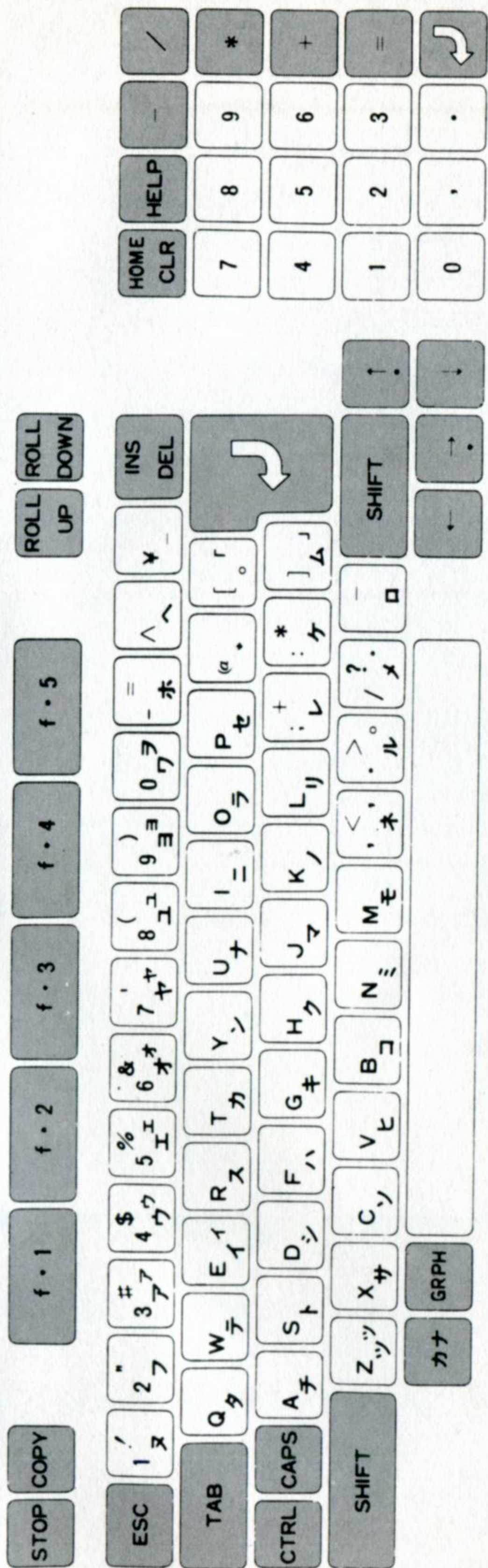
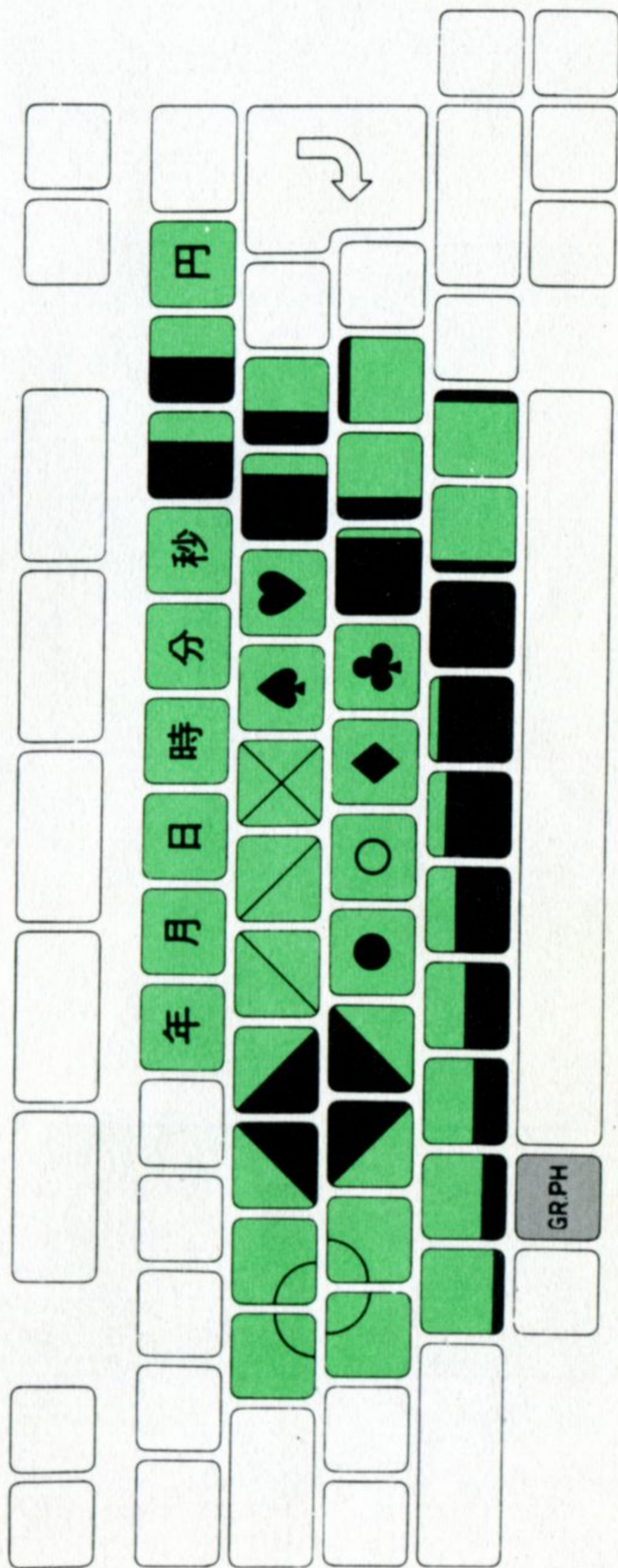
次にアセンブリ言語プログラミングをする場合には、あとで説明するモニタプログラムというものをよく使いますが、この場合、非常に便利なキーが左側のはじにあるＣＴＲＬ（コントロール）キーです。コントロールキーはＢＡＳＩＣで使っている場合と、モニタプログラムで使っている場合とでは、同じような使い方でも機能が異なりますので注意を要します。たとえば、次のようです。

組み合わせ	ＢＡＳＩＣのとき	モニタプログラムのとき
CTRL ＋B	カーソルを左へ	ＢＡＳＩＣへ戻る
CTRL ＋D	カーソルから左を消去	ディスクの内容をＣＲＴに表示
CTRL ＋R	インサートモードにする	ディスクからデータをロード

HELP キー

もう一つ面白いのは、右側のテンキーの中にあるＨＥＬＰキーです。これもＢＡＳＩＣで使うときと、モニタプログラムで使うときには意味が違います。これからアセンブラプログラムを勉強するわけですから、ＨＥＬＰは、モニタプログラムのコマンドがわからなくなったときに、コマンドの内容を教えてもらうために使うのだと覚えてください。ＨＥＬＰは CTRL ＋A でも使えます。

CTRL ＋ A

最後にＰＣ－８８０１はＩＮＳＥＲＴキーが少し変わっていて、使いにくいようです。慣れだけの問題かもしれませんが。

キーの配列

グラフィックシンボルのキー配列

1・2 PC-8801のメモリマップ

メモリマップ

PC-8801のメモリマップは大変わかりにくいもので、しばらくの間どういうふうになっているかがわからなかったのですが、最近情報がふえてやっと全体像がボンヤリ姿を現してきました。

バンク切換

PC-8801のメモリマップが複雑になっているのは、バンク切換という手法を使って、本来8ビットのパソコンのメモリ空間64キロバイトを184キロバイトにまで押し広げているからです。こうした理由から、非常に難解でわかりにくいものですが、テクニックの面白さを勉強することができます。

$0000
$4000
$8000
$8400
$C000
$FFFF

BASIC
プログラム

テキスト
ウィンドウ

メインRAM

N-BASIC
インタプリタ

N-BASIC
インタプリタ
モニタ

N88-BASIC
モニタ

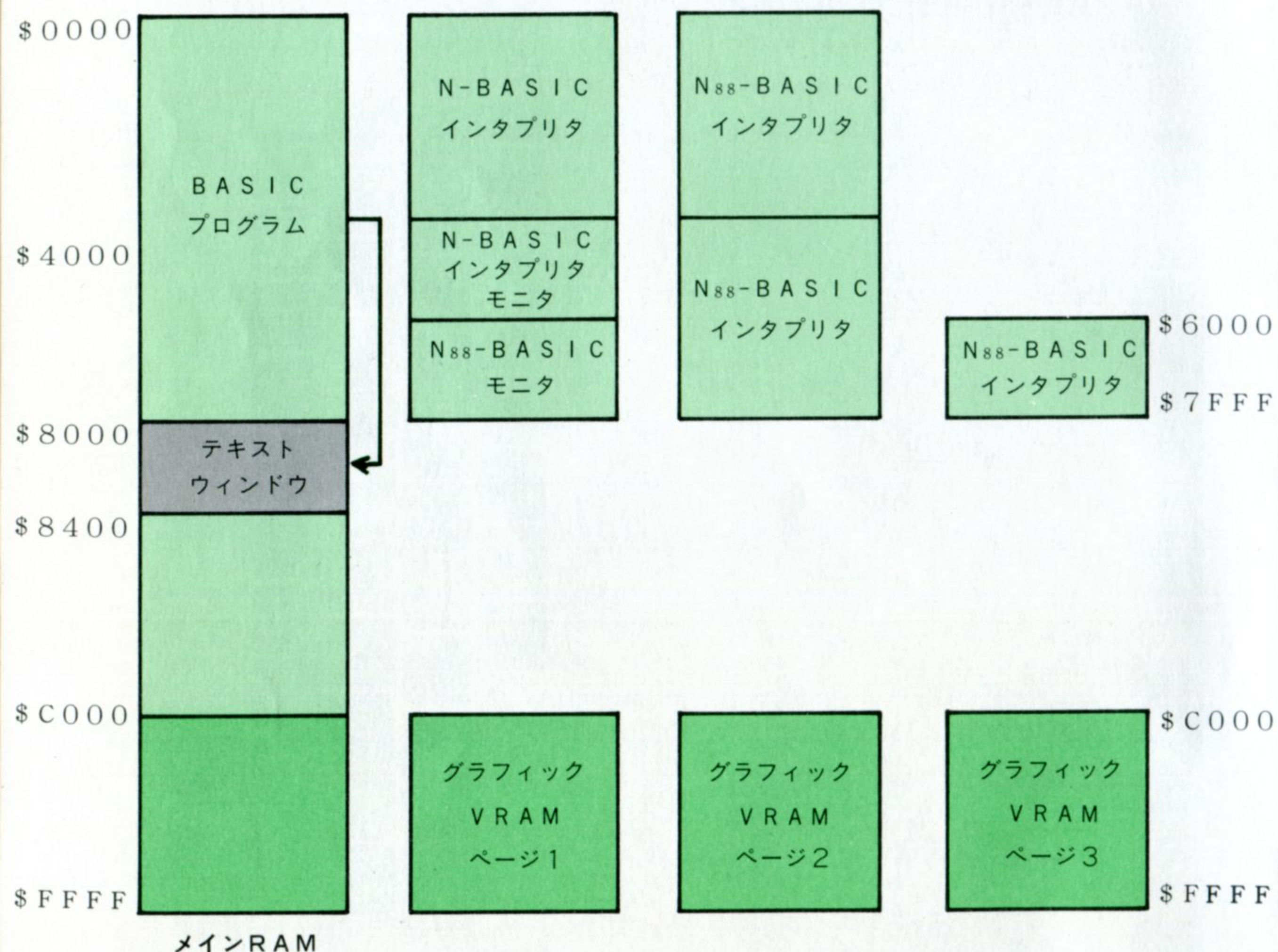
N88-BASIC
インタプリタ

N88-BASIC
インタプリタ

$6000
N88-BASIC
インタプリタ
$7FFF

$C000
グラフィック
VRAM
ページ1

グラフィック
VRAM
ページ2

グラフィック
VRAM
ページ3
$FFFF

図を見てください。何とややこしい複雑な構造となっているのでしょうか。これからアセンブリ言語のプログラムを組む上で、これがどういう関係があるのでしょうか。

まず$0000～$7FFFの範囲には、プログラムを書いてはい

けません。ここはＢＡＳＩＣのプログラムが入っているので、おかしな機械語のプログラムを書いてしまいますと、ＢＡＳＩＣにもどったときにうまく動作しなくなる心配があります。

＄８０００～＄８３ＦＦの範囲もテキストウィンドウ領域といって、機械語のプログラムを書いてはいけません。ここはＢＡＳＩＣのテキストを読み書きする窓なのです。(＄００００～＄７ＦＦＦはＢＡＳＩＣのテキストの読み出し専用で、直接ＢＡＳＩＣプログラムは書けません。)

＄Ｃ０００から＄ＦＦＦＦも本当は機械語プログラムを書かないほうがよいのです。なぜかというと、グラフィック用のＶＲＡＭがこの領域に移動することがあるので、場合によっては機械語のプログラムが破壊されてしまい、メチャクチャになる危険性が生じます。

それではいったいどこがあいているのでしょうか？　＄８４００～＄ＢＦＦＦの15キロバイトがあいているようには見えます。しかし、通常のN$_{88}$-ＢＡＳＩＣなら心配がありませんが、N$_{88}$-ＤＩＳＫ　ＢＡＳＩＣでは、＄８４００から＄ＢＯＤＡまでをジャンプテーブルに使っているので、ここも自由に使えません。

それではどこも使えないではないかといわれるかもしれません。この問題には私も相当悩まされました。妥協案として、

```
CLEAR ,&HDFFF
```

CLEAR　,&HDFFF

としておいてから、＄Ｅ０００から機械語プログヤムを入れることにしました。２キロバイトしかプログラムはかけませんが、初心者のうちはそれほど長いプログラムは書けないと思いますので、これで大丈夫と思います。

PC-9801について

ＰＣ－８８０１の後継機としてＰＣ－９８０１が出現してきています。演算スピードも大変速く、グラフィックスも相当はやいようです。だからＰＣ－８８０１をＰＣ－９８０１に変えようとする人もたくさんいるようです。けれどもＰＣ－８８０１を売ってＰＣ－９８０１にのりかえるのは損だと思います。その理由はＰＣ－８８０１はＺ80を搭載したパソコンとしては非常によくできているからで、これを半値以下で売るのは効率的でないと思うからです。できればＰＣ－８８０１はそのまま残してＰＣ－９８０１を入手する方法を考えたほうがよいと思います。アプリケーションによってはＺ80用のソフトしかない場合があり、８０８６１本に切り換えた場合にはつらいこともあると思います。

1・3 モニタプログラムの概要

これからアセンブリ言語を勉強して行く上でモニタプログラムをどう使いこなすかが大切な課題となります。一般に、パソコン用のモニタプログラムは簡単なものが多いのですが、ＰＣ-８８０１のモニタプログラムのコマンドは大変充実しています。

Ｉ／Ｏポート

とくに便利だと思うのは、ＰＣ-８００１にはなかったアセンブルやディス・アセンブル（逆アセンブルのこと）コマンドが盛り込まれている上に、プリンタの出力コマンドがモニタプログラムの中に準備されていることです。面白いと思うのはＩ／Ｏ（入出力）ポートへの入出力命令が準備されていることで、機械語で入出力を制御することがよく考えられています。また、カセットテープレコーダやフロッピーディスクへのデータの入出力命令がモニタプログラムのコマンドの中に用意されているのもよいことです。

下図にモニタプログラムのコマンドの一覧表をかかげます。

コマンド	意　味	機　能
A	アセンブル	入力した１行をアセンブルします。
B	ベース	数値の表現形式を変えます。（８進←→16進）
D	ダンプ・メモリ	メモリの内容をディスプレイに表示します。
E	エディト・メモリ	スクリーン・エディタの機能を用いてメモリの内容を変更します。
F	フィル・メモリ	メモリの内容を定数で埋めていきます。
G	ゴー	ユーザープログラムの実行をします。
I	インプット	Ｉ／Ｏポートの値を読み込みます。
L	ディス・アセンブル	機械語を逆アセンブルします。
M	ムーブ・メモリ	ある範囲のメモリの内容を他のアドレスのメモリ領域へ移します。
O	アウトプット	Ｉ／Ｏポートへデータを出力します。
P	プリンタ・スイッチ	プリンタへの出力をコントロールします。
R	リード・テープ	カセットテープからデータをロードします。
S	セット・メモリ	メモリにデータをセットします。
TM	テスト・メモリ	メモリをテストします。
V	ベリファイ・テープ	カセットテープの内容と、メモリの内容を比較します。
W	ライトテープ	メモリの内容をカセットテープにセーブします。
X	イグザミン・レジスタ	ＣＰＵのレジスタの値を調べ、変更します。

コマンド	意　味	機　能
(HELP) ＣＴＲＬ－Ａ	ヘルプ	コマンドとそのパラメータの形式をディスプレイに表示します。
ＣＴＲＬ－Ｂ	リターン	ＢＡＳＩＣモードへ復帰します。
ＣＴＲＬ－Ｄ	ダンプ・ディスク	ディスクの内容をディスプレイに表示します。
ＣＴＲＬ－Ｒ	リード・ディスク	ディスクから、データをロードします。
ＣＴＲＬ－Ｗ	ライト・ディスク	ディスクへデータをセーブします。

●「ＣＴＲＬ-」は (ＣＴＲＬ) キーを押しながら入力することを表します。

1・4 モニタプログラムの動かし方

それではさっそくモニタプログラムを動かしてみましょう。モニタプログラムを動かすにはN88-BASICかN88DISK-BASICモードで次のように入力します。

●以下、あらかじめWIDTH80，25を入力してあります。

```
How many files(0-15)? 5
NEC N-88 BASIC Version 1.0
Copyright (C) 1981 by Microsoft
55731 Bytes free
Ok
```

```
Disk version [Feb 4,1982]
How many files(0-15)? 5
NEC N-88 BASIC Version 1.0
Copyright (C) 1981 by Microsoft
44800 Bytes free
Ok
CLEAR,&HDFFF
Ok
MON

h]AE000
E000
```

CTRL +B

モニタプログラムから脱出するには、コントロールキー（CTRLキー）とBを同時におせばN88-BASICにもどります。

```
MON

h]AE000
E000   AF          XRA A
E001   32 E010     STA E010
E004   76          HLT
E005
h]^b
Ok
```

PC-8801のモニタプログラムのコマンドは22種類もあります。使うたびにマニュアルをひっくり返すのはたまりません。何かよい方法はないでしょうか。便利な方法があります。

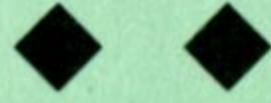
HELP キー

HELP キー（ヘルプキー）を押すか、コントロールキー（CTRLキー）とAを同時におせば、コマンドとそのパラメータの形式をCRTディスプレイ上に表示してくれるのです。

```
Monitor has following commands.
a <address>                    assemble source text lines.
bh or bq                       select radix (hexa decimal or octal).
d <start>,<end>                dump contents of memory.
e <start>                      change contents of memory by screen editor.
f <start>,<end>,<const>        fill memory by the constant.
g <start>,<break1>,<break2>    execute user program.
i <port>                       read input port.
l <start>,<end>                dis-assemble program in memory.
m <start:s>,<end:s>,<start:d>  move contents of memory block(s=src,d=dest).
o <port>,<new value>           output new value to the port.
p                              toggle print switch.
r <file name>                  read cassette file.
s <address>                    substitute memory contents.
tm                             test memory.
v <file name>                  verify cassette file.
w <file name>,<start>,<end>    write cassette file
x or x <register name>         dump all CPU registers or change the register.
CNTL-b                         return to BASIC.
h]
```

第2章

アセンブリ言語プログラミングの約束ごと

◆ ◆

2・1 いろいろな数え方(2進, 10進, 16進)

さて、コンピュータの中では数字はいろいろな表現で使われます。入門書などで2進数や16進数などの説明をお読みになった方も多いと思います。たとえば、10進数の64という数字は次のようにいろいろに表せます。

２進数の場合──→ 1000000
８進数の場合──→ 100
10進数の場合──→ 64
16進数の場合──→ 40

普通どのような数字で表されているかを明示するために、次のような記号を使います。

２進数の場合──Ｂまたは％ (BはBinaryの略)
８進数の場合──Ｏ、Ｑ、Ｃまたは＠ (OはOctalの略)
10進数の場合──Ｄ (DはDecimalの略)
16進数の場合──＆Ｈ、Ｈ、または＄ (HはHexadecimalの略)

これらの数え方のどれで表現されてもわかるようでなければいけません。もし数の変換がわからない場合には、ＢＡＳＩＣの中に便利な命令が準備されています。たとえば、10進数の100を16進数に変換する場合には、ダイレクトモードで次のように出せます。

```
PRINT  HEX$ (64)
40  ←──── 16進で答が表示される
```

10進数を８進数に変換するには次のようにします。

```
PRINT  OCT$ (64)
100
```

ＰＣ-８８００のモニタプログラムでは、16進と８進数だけが使われます。通常の場合には、16進数が使われ、８進数に切り換えるには bq と入力します。

```
mon

h]bq
q]bh
h]^b
Ok
```

Ｂは数値の基数の英語(Base) に由来しています。

数値のいろいろな数え方は、使っていくうちにだんだんわかってくるでしょう。

2・2　ニーモニックに関する注意

ニーモニック
(MNEMONICS)

ＰＣ-８８０１ではインテル形式のニーモニックが使われます。ニーモニックというのは、プログラムの命令コードに対応した暗記用のコードです。たとえば、３Ａと書くよりはＬＤＡと書いたほうが人間にはわかりやすいでしょう。ＰＣ-８８０１のＣＰＵであるＺ80の場合でもすべての命令コードにニーモニックがわり当てられています。ここでひとつの問題があります。それはＰＣ-８８０１のモニタプログラムがインテル社の形式のニーモニックを採用していることです。ところが、インテル社のニーモニックはインテル８０８０を対象にして考えられているのです。

８０８０とＺ80は同じ日本人の嶋正利氏によって設計されているので、ぜんぜん違うものではありませんが、Ｚ80は８０８０の改良形であるので、８０８０にない命令を含んでいます。そこでインテルのニーモニックを使った場合は、Ｚ80のよさを十分ひき出せない場合もあります。

絶対番地

リロケート

とくに８０８０では絶対番地方式といって、すべてのアドレス指定を絶対番地というもので書いてしまうので、メモリのある部分で書いたプログラムを他の部分へ移す（これをリロケートといいます）ことができませんが、Ｚ80の場合には、アドレス指定が柔軟になっているのでリロケートができるのです。

また困ったことに、インテルのニーモニックとザイログのニーモニックでは、ニーモニックが全く同じではありません。

たとえば同じプログラムでも次のように違ってくるのです。

```
LD   A,(E020)
CPL
LD   (E021),A
HALT
```

これは実にやっかいな問題です。しかたがありません。

そこでＰＣ-８８０１では頭をしぼって、だいたいの場合はインテル社のニーモニックで書き、インテル社のニーモニックにない相対ジャンプ命令などはザイログ社のニーモニックを使うという方式を採用しています。少し難しいかもしれませんが、このことははじめによく承知しておいてください。

2・3 ソースプログラムとオブジェクトプログラム

ソースプログラム
オブジェクトプログラム

ソースプログラムとオブジェクトプログラムという名前は難しそうですね。ソースというのはお料理に使うソースではなくて、源くらいの意味です。ソースプログラムは原始プログラムと訳します。オブジェクトというのは、対象とか目的とかいう意味で、オブジェクトプログラムは目的プログラムと訳します。ソースプログラムはニーモニックを使ってアセンブリ言語で書かれたもので、オブジェクトプログラムは0と1だけの機械語で書かれたプログラムです。

ソースプログラムをオブジェクトプログラムに変換するプログラムのことをアセンブラプログラムといいます。ソースプログラムからオブジェクトプログラムへの変換を、プログラムでなく人間が表を参照しながら実行することも考えられます。このような場合、少し気どってハンドアセンブリといいますが、大変な作業です。

ハンドアセンブリ

ハンドアセンブリの場合でも、アセンブラプログラムによるアセンブリ作業でも、直接0と1の機械語におとすことはまれです。普通は16進数で書かれたプログラムにおとします。16進数で書かれたプログラムを機械語におとすためのプログラムもあって、ヘキサデシマルローダ (Hexadecimal loader)といいます。一般にはヘキサデシマルローダのことは意識しません。

ヘキサデシマルローダ

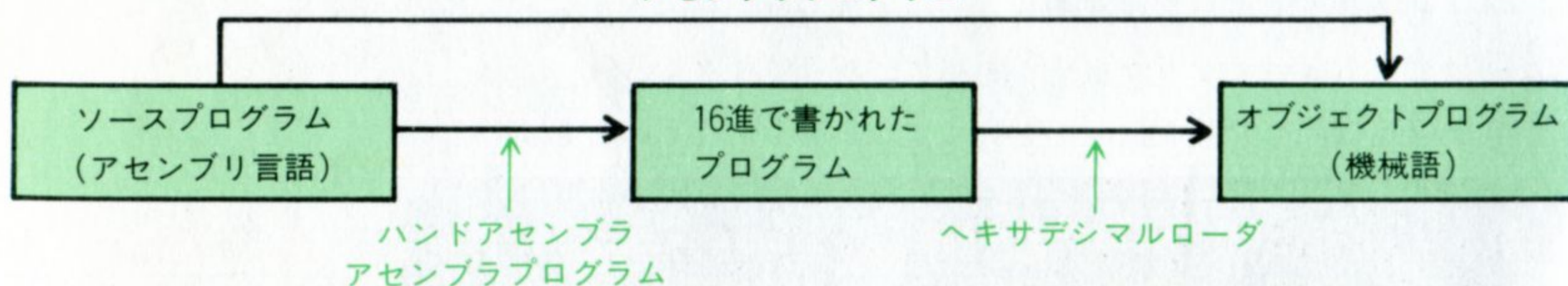
アセンブラプログラム

ソースプログラム
（アセンブリ言語）

16進で書かれた
プログラム

オブジェクトプログラム
（機械語）

ハンドアセンブラ
アセンブラプログラム

ヘキサデシマルローダ

ＰＣ-８８０１の場合にはハンドアセンブルをする必要がありません。モニタプログラムの中にはアセンブラの機能がはいっているからです。これを有効に使えば、恐るべきハンドアセンブリの作業から解放されます。あなたがすることはどのようなプログラムを作るかを考え、アセンブリ言語で書くまでであって、そのあとはＰＣ-８８０１にまかせればよいのです。便利になったものですね。

2・4 アセンブリ言語プログラムの書き方

アセンブリ言語でプログラムを書くうえで、いくつかの約束があります。

まず、前に示したアセンブリ言語プログラムのコーディングシートを見てください。いくつかの欄（コラム）にわかれていますね。最初にラベル（label）という欄があります。次にニーモニック（またはオペコード）という欄があります。さらにオペランドの欄があります。それからいくつかの欄がならび、注釈（コメント）となっています。オペコードまたはニーモニックの欄は決して空白にはなりません。ニーモニックか擬似命令というものがはいります。オペランドまたはアドレスの欄はアドレスまたはデータが書かれますが、空白のこともあります。注釈の欄はプログラムの内容を説明するところです。注釈の書き方にも作法があるようですが、それはあとで説明します。当分は自分でわかりやすいように使ってください。

ラベルはプログラムをわかりやすくするためのもので、自分で勝手につけるものですが、でたらめにつけると混乱がおきます。ラベルのつけ方に関する注意は次のようです。

❶ ニーモニックとまぎらわしいラベルは使わない。

❷ 長すぎる名前は使わない。

❸ 特殊な記号や小文字は使わない。

❹ 数字からはじまるラベルは使わない。文字ではじめる。

❺ 一部だけちがうようなラベルはまぎらわしいから使わない。

デリミタ

次に、アセンブラで使われるデリミタについて説明します。直訳すればデリミタは分離記号ですが、

コロン（：）　空白（␣）　カンマ（，）　セミコロン（；）

の使い方はやかましい規則があります。コロン１つやコンマの使い方くらいでプログラムが動かないのはかなわないので、あやしいと思ったらよけいな空白をいれたり、セミコロンやコロンは使わないことです。ＰＣ-８８０１のモニタプログラムのアセンブラでは、標準的な使い方さえしていれば安全です。あまりきざにしなければ安全だということです。

2・5 擬似命令

擬似命令
(PSEUDO OPERATION)

本式のアセンブリ言語で書かれたプログラムを見るとき、ドキッとさせられるのは、擬似命令とよばれるものの存在です。

普通よく使われるものとしては、つぎのものがあります。

DATA
EQUATEまたはDEFINE
ORIGIN
RESERVE

アセンブリ言語の命令としては、機械語には全く翻訳されないのですが、アセンブラプログラムに指示を与えるものがあります。これが擬似命令(PSEUDO-OPERATIONS)で、上にあげたものがその代表例です。擬似命令はオペコードもしくは同じことですがニーモニックの欄に書き、アドレスやデータが必要な場合にはオペランドの欄にそれを書きます。

DATA擬似命令

DATA擬似命令はメモリの中にデータをいれるためのものです。次のようにな書式にします。

```
KEY        DATA        'JISC6226'
A          DATA        B0A1,B0A2,B0A4
```

EQUATE擬似命令

EQUATE(またはDEFINE)命令はアドレスやデータにラベルをわり当てるためのものです。この擬似命令は次のようにして使います。擬似命令はメモリの中にデータを書き込むのではなくて、アセンブラプログラムが作る記号表の中にラベル名を登録するだけのところがDATA文とは違います。EQUATE命令はプログラムの先頭にいれたほうがいいと思います。

```
CRTCP      EQU         50
CRTCC      EQU         51
DISPC      EQU         52
DISPR      EQU         53
```

ORIGIN擬似命令

ORIGIN命令はあるプログラムや、サブルーチンがメモリのどこに位置するかを示します。あいまいさをなくすためにもORIGIN命令を使ったほうがよいでしょう。

```
           ORG         E000
```

RESERVE擬似命令

RESERVE命令はメモリの中にある領域を確保しておくためのものです。あまり使わないと思いますが、次のようにして使います。

```
CODET      RESERVE     E100
```

2・6 ザイログZ80アセンブラの約束ごと

ＰＣ-８８０１では、インテル社のアセンブラの記法を使いますが、もともとＰＣ-８８０１はZ80をつんだパーソナルコンピュータなのですから、ザイログ社のアセンブラの約束を憶えておくのも悪くないことだと思います。

まずデリミタの使い方ですが、ラベルのあとには原則としてコロンをおきます。擬似命令のあとにはコロンはおきません。

ニーモニックまたはオペコードのあとには、スペース（空白）をおきます。空白とは何もおかないのではありません。スペースバーという長いキーを１回たたくのです。

オペランドの間には必ずコンマをうちます。注釈（コメント）の前にはセミコロン(;)をつけます。ある番地を参照するときには、その番地の両側にカッコをつけます。

次にラベルですが、これは６文字以内ということになっています。最初の文字は文字でなければならないことはもちろんです。数字を先頭にもってきてはいけません。さらにラベルは大文字で書いたほうがよいと思います。Z80ＣＰＵのレジスタ名などをラベルに使ってはいけないことはもちろんです。

擬似命令については次のようなものがあります。

ＤＥＦＢ――ＤＥＦＩＮＥ　ＢＹＴＥ
ＤＥＦＬ――ＤＥＦＩＮＥ　ＬＡＢＥＬ
ＤＥＦＭ――ＤＥＦＩＮＥ　ＳＴＲＩＮＧ
ＤＥＦＳ――ＤＥＦＩＮＥ　ＳＴＯＲＡＧＥ
ＤＥＦＷ――ＤＥＦＩＮＥ　ＷＯＲＤ
ＥＮＤ　――ＥＮＤ
ＥＱＵ　――ＥＱＵＡＴＥ
ＯＲＧ　――ＯＲＩＧＩＮ

2・7 8080/8085アセンブラの利点

ＰＣ-８００１はZ80というマイクロプロセッサを搭載しながら、なぜインテル８０８０や８０８５用のアセンブラを採用しているのでしょうか。しかもインテル社のアセンブラではＺ80の機能を十分に引き出すためには、いろいろつぎはぎ的な工夫をしなければななりません。

日本電気がインテル社のアセンブラを採用した本当の理由はよくわかりませんが、私はＣＰ／Ｍ（シーピーエム）というオペレーティングシステムを意識してのことだろうと思います。

ＣＰ／Ｍ

ＣＰ／Ｍというのは、ディジタル・リサーチ社という米国のソフトウェア会社が、インテル社の開発システム（ＭＤＳと略称することがあります）で動くように作ったオペレーティングシステムです（ただし、現在はＭＤＳでは動きません)。 Control Program for Micro-computor の頭文字をとってＣＰ／Ｍと略称します。

ＭＤＳ

いったい、なぜＣＰ／Ｍがそんなに大切なのでしょうか。勉強がすすんで行けばよくわかりますが、この本で勉強するようなアセンブラをあやつるのは勉強のためには非常に手軽でよいのですが、キチンとした仕事をするには厄介です。とくにパソコンと外部の周辺機器、たとえばフロッピーディスクやプリンタを動かしたり、あるプログラムとあるプログラムを結合させたり、キーボードから入力されたデータによってプログラムを動かしたりすることを考えてみますと、大変面倒なものになってしまいます。そこで、こうした面倒な仕事は全部やってくれるプログラムがあると便利です。言ってみればプログラムを管理したり制御するプログラムで、これをオペレーティングシステムと呼んでいます。（O．S． オーエスと呼ぶことがあります。)

ＯＳ

こうしたＯＳはいろいろ提案されていますが、結局ディジタル・リサーチ社のＣＰ／Ｍが市場で一番大きいシェアを占めてしまいました。つまり一番多くの人に使われるようになったということです。こうなると恐ろしいもので、多くのソフトウェア会社がなだれをうってＣＰ／Ｍの下で動くプログラムを作り始めました。パソコンはソフトウェアがなければただの箱にすぎません。従ってパソコンを利用する人はソフトウェアが一番多いＣＰ／Ｍに注目するようになったのです。

ＣＰ／ＭをＺ80用に書き直せばよいという意見もあるかもしれませんが、そうするといろいろ手直しが必要で、せっかくのソフトウェアの財産が利用できなくなったりします。技術の世界も理論通りすっきり、というわけには行かないのです。

第3章

まずはやさしいプログラムから

◆◆◆

3

3・1　1と0をひっくり返すこと

まず、やさしいプログラムを手がけることにしましょう。始めのうちは理屈よりもキーをたたいて、体で覚えるようにしてください。ある程度練習量をこなしたら、どうしてだろうかと考えてみてください。

最初に、CLEAR ␣, &HDFFF ⏎とし、次にPC-8801のモニタプログラムを呼び出します。

MON ⏎　　とするか　mon ⏎

とすれば、画面に次のような表示がでます。

```
MON

h]■
```

アセンブリ言語のプログラムを作るためには、どこかにプログラムを格納してやらなければなりません。この目的のためにPC-8801のメモリがどう使われているかを見てみますと、ビッシリ詰まっていて、なかなかあいている所がありません。だいたい大丈夫そうなのが、$C000番地から$E5FF番地です。ここでは、それほど長いプログラムを作るつもりはないので$E000番地からプログラムを格納することにします。

Aコマンド

モニタプログラムの中にはアセンブル機能を持ったAコマンドが用意されていて、インテル形式のアセンブリ言語プログラムを機械語プログラムに変換してくれます。Aコマンドの使い方は、

入力形式 ： A〔格納開始番地〕

です。ここでは入力形式は16進のまま行ないますので、とくに指定はしません。さらに、機械語プログラムの格納開始番地は$E000番地からにします。このため次のように入力します。

```
h]AE000 ⏎
```

こうしますと、画面には次のような表示がでて入力待ちになります。

```
MON

h]AE000
E000                    ■ ←――カーソルがここで
                        ↑     ブリンクしている
```

ここから、L D A スペース E 0 0 0 ⏎
の順にキーインする

そこでアセンブリ言語で書かれたプログラムを１行ずつ入力してやることにします。

```
h]AE000
E000                LDA E010
```

スペースを１つあける

Ｅ０１０まで打ったら ⏎ を押しますと、あっという間にアセンブルが終了します。

```
MON

h]AE000
E000    3A E010       LDA E010
E003                  ■
```

カーソルがここでブリンク

とても簡単なのには驚かれるでしょう。

こうして次々に入力していきます。

```
h]AE000
E000    3A E010       LDA E010
E003    2F            CMA
E004    32 E011       STA E011
E007    76            HLT
E008                  ■
```

⏎キーを押す

ＨＬＴ（停止）命令を入れてリターン ⏎ を押すと＄Ｅ００８番地になりますが、入力すべきプログラムは終了しているので、何も打たずにリターン ⏎ だけをもう一度押すと再びｈ］の表示が出て、アセンブラＡのモードから脱出できます。

画面ではうまく入ったようでも、メモリには違うものが入っているのはよくあることですから、逆アセンブラのＬコマンドを使って確認をしておきます。

Ｌコマンド

```
h ] LE000, E007 ⏎
```

これは＄Ｅ０００番地から、＄Ｅ００７番地の内容を読み出して機械語プログラムからアセンブリ言語プログラムへ逆アセンブルしなさいということです。結果は次のようになります。

```
h ] LE000, E007
E000    3A E010        LDA E010
E003    2F             CMA
E004    32 E011        STA E011
E007    76             HLT
h ] █
```

確かに正しく入力されていましたね。

Ｓコマンド

これでプログラムは入ったのですが、データが入っていませんのでデータをしまいます。プログラムの意味は、＄Ｅ０１０番地の中味をアキュムレータに呼び出し、０と１を入れかえて再び＄Ｅ０１１番地にしまいなさいということです。従って、＄Ｅ０１０番地には＄００をしまっておきます。このためには、Ｓコマンド（セット・メモリ・コマンド）を使います。

```
h ] SE010  ←――――――― ⏎キーを押す
E010 00- ␣␣
          ↑ ――――――― ここにデータを入れる
     ↑ ―――――――――― Ｅ０１０番地の現在の中味
```

ハイフンがでてきますから、００と打って、スペースバーという長い棒状のキーを押します。⏎キーを押してはいけません。悪いことはないのですがｈ］にもどってしまって、連続してデータを打ち込むことができなくなります。もしそうしてしまったら、ｈ］ＳＥ０１１ ⏎として続けてやればよいのです。＄Ｅ０１１番地にも＄００を入れておきます。

```
h ] SE010           ↓ ――― 変更前のＥ０１１番地の中味
E010 00-00 FF-00    ⏎キーを押す
              ↑ ――― Ｅ０１１番地の中味を００とする
         ↑ ―――――― ００と打ってスペースバーを押す
```

００を入力したら ⏎ を押します。こうしないとＳコマンドのモードから脱けられません。⏎ を押せばＳコマンドのモードから脱けだせて再びｈ］の表示がでます。

次にすることはメモリの内容を見ておくことです。メモリに正しく入っていない場合も十分考えられますから、必ず確認します。これはDコマンド（ダンプ・メモリ・コマンド）で実行できます。いま、$Ｅ０００番地から$Ｅ０１１番地までの中味を見ればよいわけですから、次のようにします。

Dコマンド

```
h]DE000, E011
E000 3A 10 E0 2F 32 11 E0 76 00 FF 00 FF 00 FF 00 FF
E010 00 00
```

これも問題ありませんでしたね。そこで、いよいよプログラムを走らせます。これはきわめて簡単でプログラムの開始番地に制御を移してやればよいのです。これにはGコマンド(ゴー・コマンド)を使います。

Gコマンド

```
h]GE000 ⏎
```

一瞬のうちにプログラムは終了し、ｈ］の表示がでます。そこで、メモリの中をのぞいてやり、プログラムが正しく実行されたかどうか調べてやります。

```
h]DE000, E011
E000 3A 10 E0 2F 32 11 E0 76 00 FF 00 FF 00 FF 00 FF
E010 00 FF
```

Ｅ０１１番地の中味がＦＦになっている

Ｅ０１０番地の中味

$E011番地の中味がＦＦに変化しています。これで正しく実行されたことがわかるのです。キツネにつままれたような顔をしないでください。下の図を見ればわかるでしょう。

〈$Ｅ０１０番地の中味〉

16進法の００ ⟶ ２進法では　０００００　００００

↓ ０と１をひっくり返す

16進法のＦＦ ⟵ ２進法では　１１１１　１１１１

〈$Ｅ０１１番地にしまわれる〉

一番簡単なプログラムでしたが、プログラムはプログラムです。

かなりいろいろなことがでてきました。$Ｅ０１０番地の中味を変えていろいろ実験してみてください。

このプログラムにでてきたアセンブリ言語の命令コードについてやさしく説明しておきましょう。

２進数⟷16進数変換表

２進数	16進数
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

LDA
(LOAD ACCUMULATOR FROM MEMORY USING DIRECT ADDRSSING)

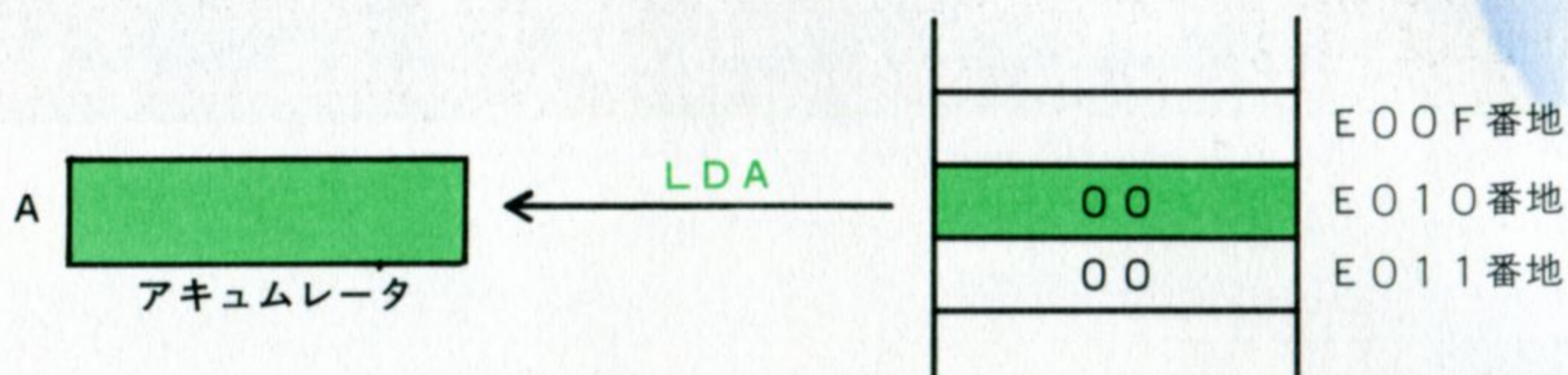
LDA

●メモリの中にしまわれているデータをアキュムレータというレジスタに呼び出してくるものです。

●プログラムの中では次のように使われていました。

LDA E010

●注意すべきことは$E010というデータをアキュムレータに呼び出してくるのではなく、$E010番地の中味をアキュムレータに呼び出してくることです。

		番地
		E00F番地
A (アキュムレータ) ← LDA	00	E010番地
	00	E011番地

●機械語プログラムでは、

3A 10E0

と交換され、番地の上位と下位が逆転していることに注意してください。(逆アセンブル結果でなく、メモリの中味を見てください)

CMA
(COMPLEMENT ACCUMULATOR)

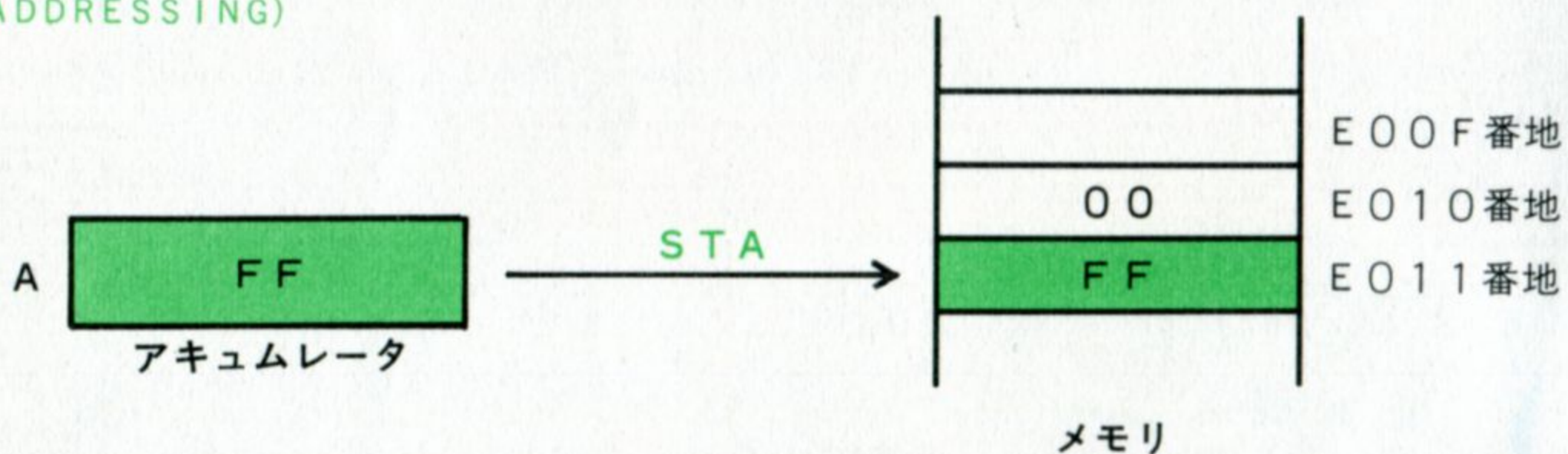
CMA

●アキュムレータの中味をひっくり返すものです。0を1に1を0にします。

STA
(STORE ACCUMULATOR IN MEMORY USING DIRECT ADDRESSING)

STA

●アキュムレータの中味をメモリの中にしまいます。STAはLDAの逆の働きをしています。

	メモリ	番地
		E00F番地
	00	E010番地
A FF (アキュムレータ) → STA	FF	E011番地

●プログラムの中では次のように使われていました。

STA E011

●機械語プログラムでは、

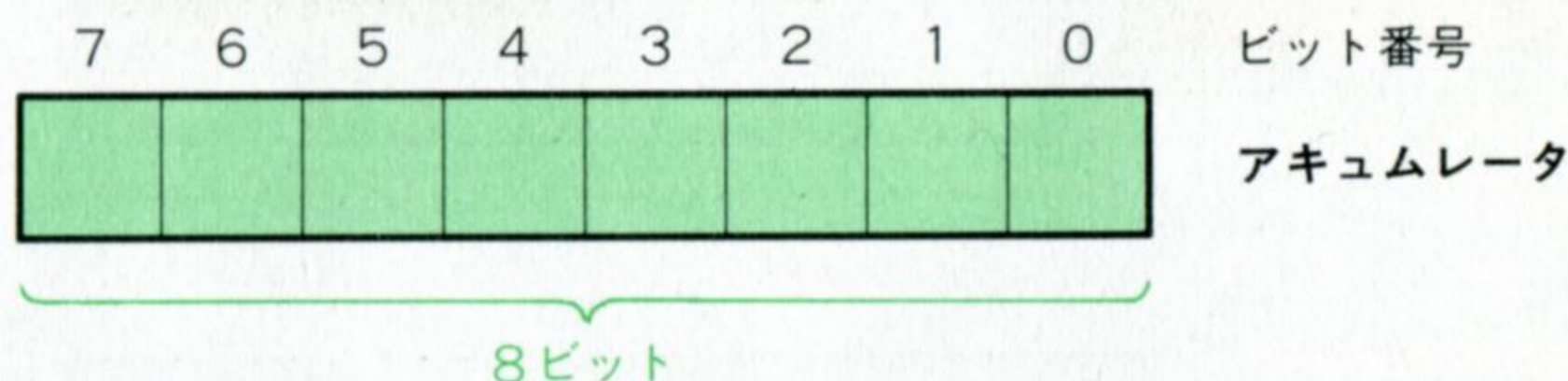
32 11E0

と変換されています。

3・2 左へシフトすること

最初のうちはなるべく簡単な命令で、なるべく短いプログラムで勉強することにしましょう。基礎をしっかり勉強することが必要です。コンピュータでよく使われるテクニックにシフトのテクニックがあります。本節ではこれを勉強することにします。

さて、どんなマイクロプロセッサでも必ずアキュムレータ(Accumulator)というものを持っています。他のものを省いても、これだけは省略できません。

7	6	5	4	3	2	1	0	ビット番号
								アキュムレータ

8ビット

PC-8801のようにZ80マイクロプロセッサを使っている場合には、8ビットのアキュムレータがあります。これがインテルi8086のように16ビット・マイクロプロセッサですと、16ビットのアキュムレータを持ちます。

アキュムレータの中では、データは2進数で格納されます。たとえば、10進数の4ですと、

$$4=1\times2^2+0\times2^1+0\times2^0$$

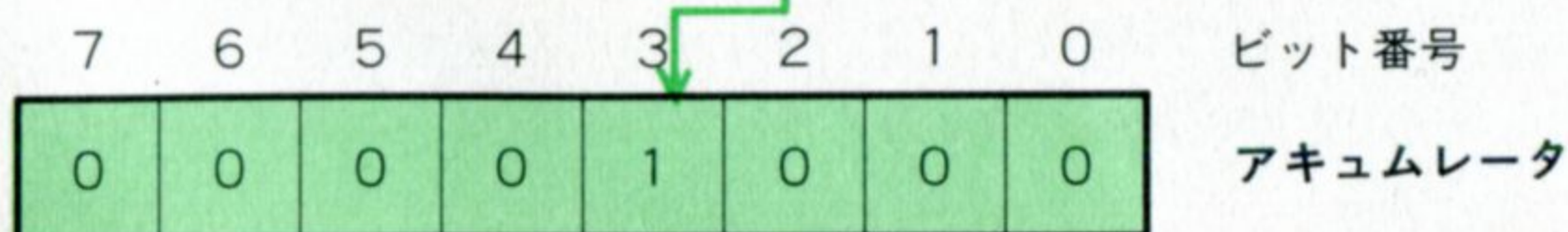
でしたから、2進数で表現しますと、0000　0100となります。これをアキュムレータに格納してみましょう。

7	6	5	4	3	2	1	0	ビット番号
0	0	0	0	0	1	0	0	**アキュムレータ**

次に10進数の8を2進数で表現してみましょう。

$$8=1\times2^3+0\times2^2+0\times2^1+0\times2^0$$

でしたから、2進数で表現しますと0000　1000となります。これをアキュムレータに格納してみますと次のようになります。

7	6	5	4	3	2	1	0	ビット番号
0	0	0	0	1	0	0	0	**アキュムレータ**

よく見くらべてください。画白いことに気がつきませんか。8は4の2倍ですが、2倍するとビットが1つ左にシフトしているのです。これは2進法の性質で、どんな数でやっても同じようになります。すなわち、2を掛けることは左へ1ビットずつシフトすることになるの

です。

●$2^0=1$と約束されています。

もう1つ例をやっておきましょう。今度は10進数の7を例にとってみます。

$$7=1\times2^2+1\times2^1+1\times2^0$$

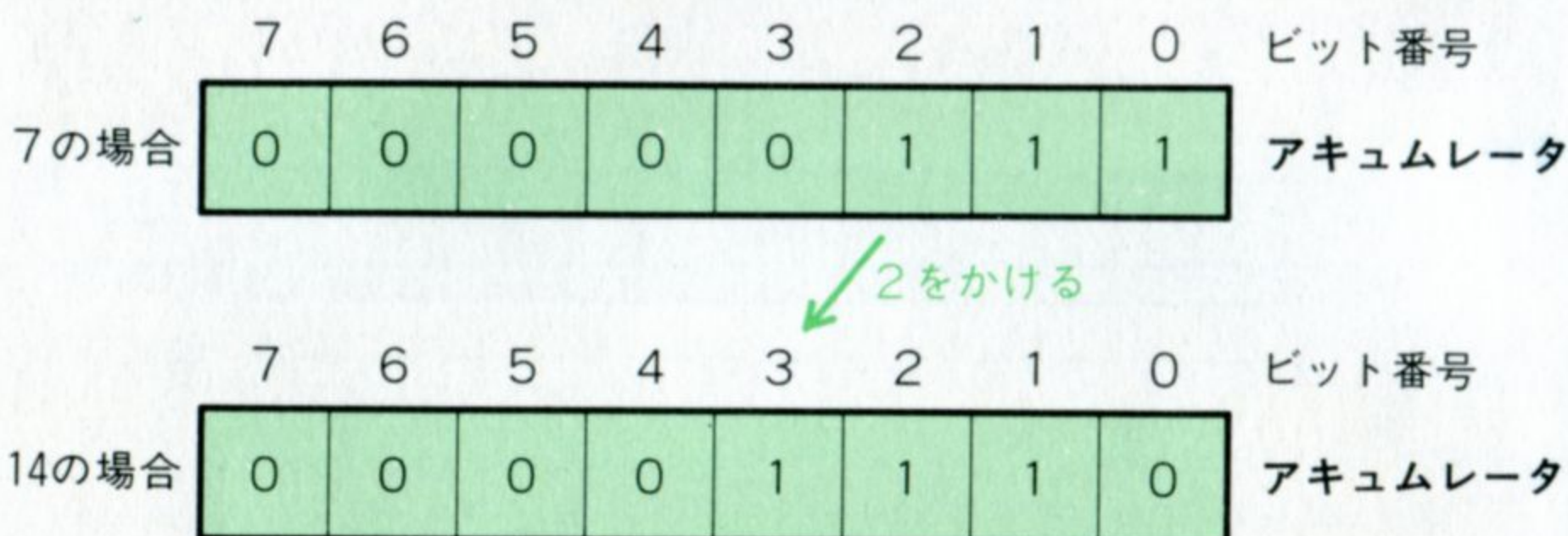
2進数で表現すると、0000　0111です。一方、10進の14を調べてみますと、

$$14=1\times2^3+1\times2^2+1\times2^1+0\times2^0$$

ですから、2進数で表現すると、0000　1110です。この2つの数字のアキュムレータへの格納状態を見てみましょう。

	7	6	5	4	3	2	1	0	ビット番号
7の場合	0	0	0	0	0	1	1	1	**アキュムレータ**

↙2をかける

	7	6	5	4	3	2	1	0	ビット番号
14の場合	0	0	0	0	1	1	1	0	**アキュムレータ**

ジッと眺めていると、いろいろなことに気がついてくるでしょう。2を掛けることは上の図で、色アミの部分が左へ1ビットずつシフトしているだけだとわかります。しかし、はみ出してしまう先頭の0はどうしたらよいでしょう。これは下の図の第0ビット、つまりお尻につければよいことがわかります。本格的なことは掛け算のところでお話しますが、大雑把にいうと、

2をかけること　=　左へ1ビットずつシフト

であることがわかるでしょう。

そこで、これを逆に読むと左へ1ビットずつシフトするということは、2を掛けることに相当します。最初から掛け算をするのは大変なので、同じ数を加えあわせると2倍になることを利用してシフトをさせてみましょう。次のことを思い出して下さい。

A＋A＝2A

以上で準備終了です。次に実際にプログラムを組んでみましょう。

まずBASICモードで、

CLEAR ␣,&HDFFF ⏎

としてください。こうしておかないとBASICモードにもどったとき、暴走のおそれがあります。

次にMON ⏎ と入力します。すると1行とばして画面に、

h]

と表示されますから、AE000 ⏎ と入力します。これで$E000番地からアセンブリ言語のプログラムを入力することができるようになりました。カーソルは$E000番地の表示から少し離れた位置に移動していますから、次のように入力してやります。

```
E000                    LDA E010 ⏎
```

●入力を間違えたときは、間違えたところの区切りの番地から、たとえばAE004 ⏎ などと入力してやる。

すると画面は次のように変化しているはずです。

```
E000   3A E010      LDA E010
E003                █
```

カーソルはここで点滅

そこで、さらに入力をくり返していきますと、図のようになるはずです。

```
CLEAR ,&HDFFF
Ok
MON

h]AE000
E000   3A E010      LDA  E010
E003   87           ADD  A
E004   32 E011      STA  E011
E007   76           HLT
E008                █
```

カーソル

HLTまで入力が終了したら、新しいアドレスが出たところで、⏎ キーを入力するか、STOP キーもしくは CTRL +C を入力してアセンブルモードから脱出します。

アセンブルが終了したら、必ずやっておくべきことがあります。それは逆アセンブル操作です。自分では正しく入力したつもりでも、正しく入っていないということはよくあることです。そこで逆アセンブルにかけておきます。いま画面は、

h]

となっていますから、続けて、

```
h]LE000, E007 ⏎
```

と入力します。すると画面上に、$E000番地から$E007番地までの逆アセンブル出力がうつります。

```
h]LE000, E007
E000   3A E010    LDA  E010
E003   87         ADD  A
E004   32 E011    STA  E011
E007   76         HLT
```

逆アセンブル操作で正しいプログラムが入っていることを確認したら、次のステップへすすみます。

E000	機械語プログラム
E007	
	未使用
E010	データ格納
E011	結果格納
	未使用
E5FF	

メモリマップ

データ格納領域と結果の格納領域を決めてやらなければならないのです。データの格納領域はどこでもいいのですが、＄Ｅ０１０番地とし、＄０１を書いておくことにし、結果の格納領域は＄Ｅ０１１番地とします。ここは＄００にリセットしておきましょう。そうしないと本当に動作しているのかどうか判定がつきません。

そこで、

h]SE010 ⏎

としてやります。すると画面は次のようになるはずです。

```
h]SE010
E010 FF-
```

＄Ｅ０１０番地にはＦＦというデータが入っているのですが、これを＄０１に変更したいと思います。そこで０１を入力して、**スペース・バーを押します**。⏎を押してしまうと、ｈ］のコマンド待ちになってしまいます。そうなってもかまわないのですが、連続して変更していきたいので、スペースバーの方がよいのです。図のようにしてください。

```
h]SE010
E010 FF-01 FF-00 ⏎
```

これでプログラムを動かす準備ができました。あとはプログラムを走らせるだけです。

```
h]GE000 ←———— ⏎キーを押す
h]
```

●失敗した場合はもう一度最初から。くじけずに！

一瞬のうちにｈ］が画面に出たと思います。出なければ暴走で失敗です。STOPを押したままリセットをかけるか、リセットをかけるほかありません。ここがＢＡＳＩＣと違うつらいところです。普通にやれば大丈夫と思います。

さてこれでプログラムは正しく実行されているかどうか調べてみ

ます。メモリの内容をダンプさせてみればよいのです。

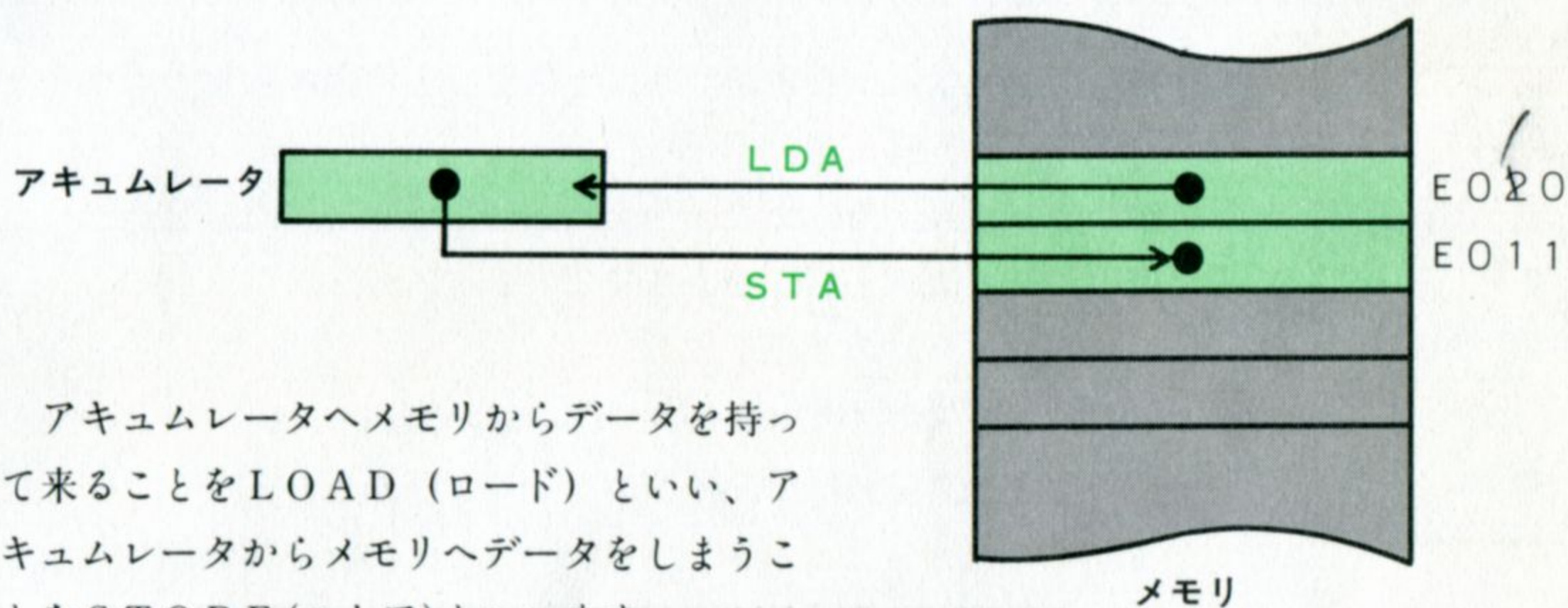
```
h]DE000, E011
```

こうしますと、画面にメモリの＄E０００番地から＄E０１１番地の内容が表示されます。

```
h]DE000, E011
E000 3A 10 E0 87 32 11 E0 76 76 FF 00 FF 00 FF 00 FF
E010 01 02
      ↑
      └────────SE011番地
```

＄E０１１番地が０２に変化していますので正しく実行されました。

さてプログラムの内容を説明しておきましょう。

アキュムレータ
LDA
STA
E010
E011
メモリ

アキュムレータへメモリからデータを持って来ることをLOAD（ロード）といい、アキュムレータからメモリへデータをしまうことをSTORE(ストア)といいます。

LOADやSTOREには同じようなものがたくさんあるのですが、LDAやSTAは最も基本的なもので、メモリの番地を直接指定するものです。

　LDA　E010

は＄E０１０番地の中味をアキュムレータにロードするもので、

　STA　E010

はアキュムレータの中味をメモリの＄E０１１番地にストアします。

ADDはレジスタやメモリの中味をアキュムレータに加えるものです。いまは、

　ADD　A

ですから、アキュムレータの中味をアキュムレータに加えており、２倍していることになります。

3・3 マスクをかけること

ANI
(AND IMMEDIATE
WITH ACCUMULATOR)

マスクをかけるというテクニックは、マイコンの世界では実によく使われます。これからも、しばしば登場すると思います。簡単にいうと、ある部分をかくしてしまうことをマスクといいます。それでは、コンピュータでやるにはどうしたらよいでしょうか。命令を覚えながら進めてゆくことにしましょう。

AND

X	Y	XANDY
0	0	0
0	1	0
1	0	0
1	1	1

OR

X	Y	XORY
0	0	0
0	1	1
1	0	1
1	1	1

ANIは、ANIに続くデータとアキュムレータの内容とのANDをとるものです。AND（アンド）とかOR（オア）の論理演算についてはなんの入門書にも書いてありますので、くだくだしく説明はしないことにします。要するに1を真（TRUE）、0を偽（FALSE）としますと、両方とも真であるときに真となるのがANDで、片方でも真であれば真というのがORです。

ANDの説明は以上ですが、ANIの働きを解説するために、

ANI　F0

を例にとって考えてみましょう。16進法でいう＄F0は2進法に直して表現してみますと、次のようになります。

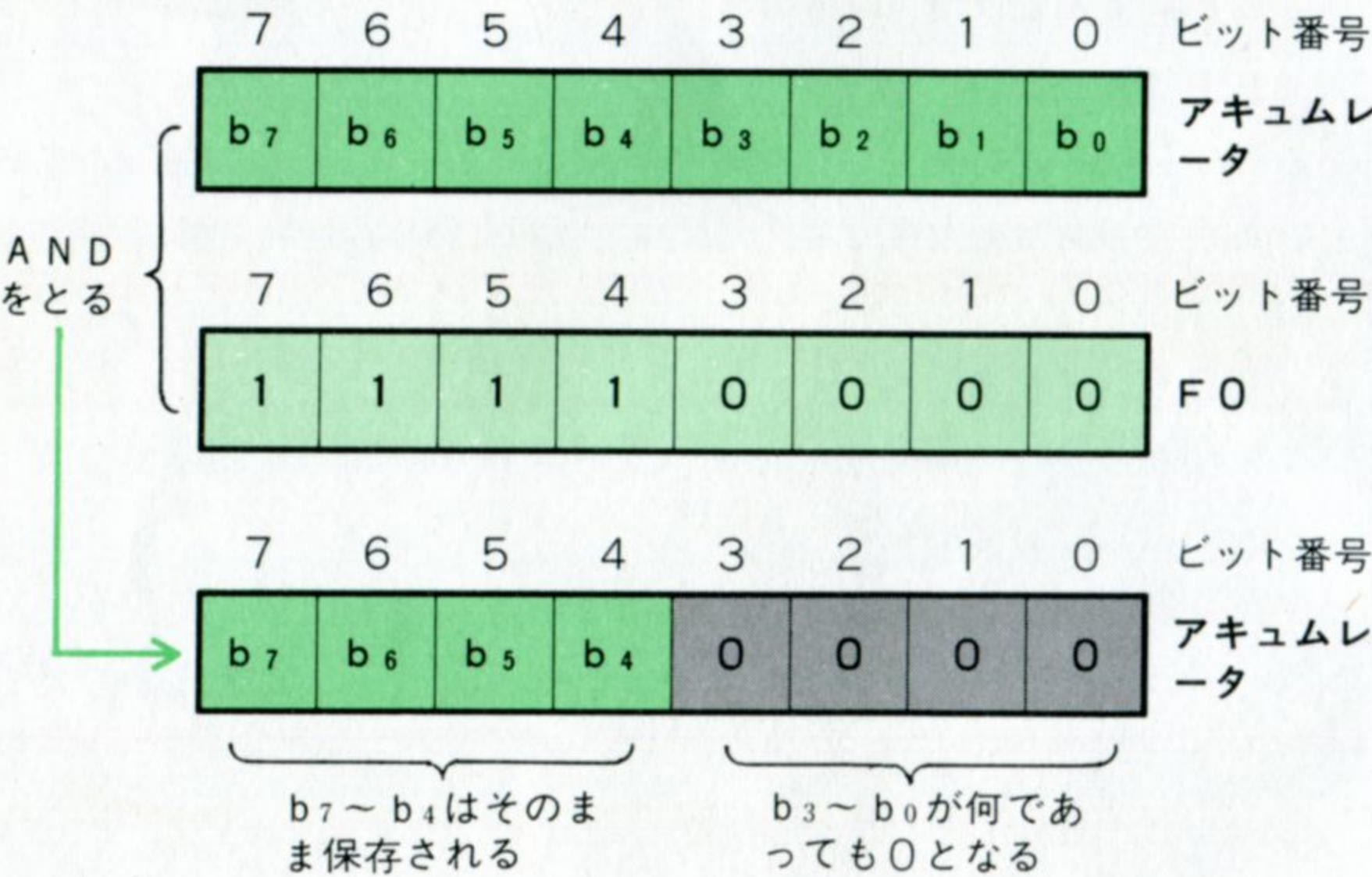
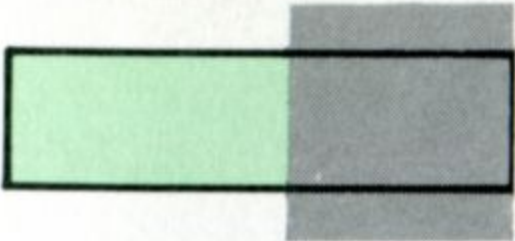
$(F0)_{16} = (11110000)_2$

<添字の16は16進法を表し、2は2進法を表します>

そこで、ANIの動作を図解すると下のようになります。

	7	6	5	4	3	2	1	0	ビット番号
	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	アキュムレータ
ANDをとる	7	6	5	4	3	2	1	0	ビット番号
	1	1	1	1	0	0	0	0	F0
	7	6	5	4	3	2	1	0	ビット番号
→	b_7	b_6	b_5	b_4	0	0	0	0	アキュムレータ

b_7～b_4はそのまま保存される　　b_3～b_0が何であっても0となる

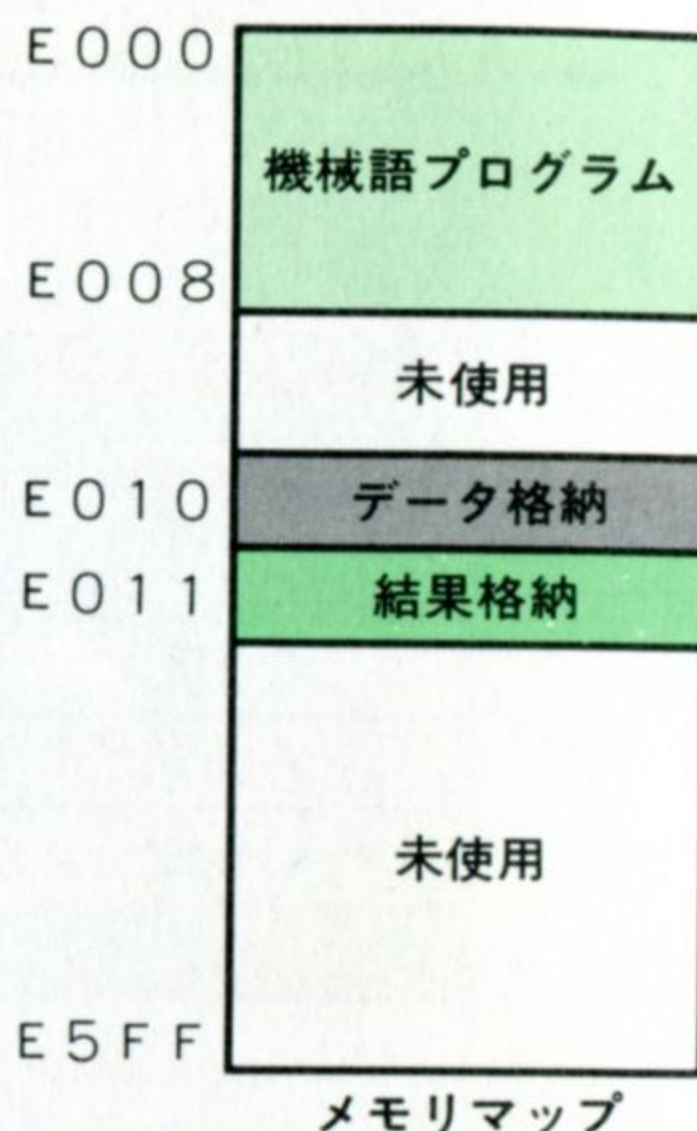
ちょうどマスクをかけたようになる

つまり、アキュムレータの最初の半分の中味はそのまま保存され、後半の中味はマスクされたように見えなくなってしまいます。

こうしてANIを使うとマスクのテクニックが使えることになりました。

それではさっそくモニタモードにするようにMON ⏎ を押して、次のプログラムを入れてください。

```
E000    3A E010       LDA   E010
E003    E6 F0         ANI   F0
E005    32 E011       STA   E011
E008    76            HLT
```

番地	内容
E000	機械語プログラム
E008	未使用
E010	データ格納
E011	結果格納
	未使用
E5FF	

メモリマップ

プログラムを入れ終わったら必ず逆アセンブルをしてください。

h]LE000, E008 ⏎ でよかったですね。

次にデータ$FFを$E010番地に、結果の格納領域として$E011番地に$00を入れておいてください。どうしたか覚えていますね。

h]SE011 ⏎

としてから必要なデータを入力していきます。

ここで一応メモリの中味を見ておきましょう。

```
E000 3A 10 E0 E6 F0 32 11 E0 76 FF 00 FF 00 FF 00 FF
E010 FF 00
```

これらが終わったら、

h]GE000 ⏎

でプログラムを走らせます。

すべてが終わったら、Dコマンドで結果を確認します。

h]DE000, E011 ⏎

うまくいっていますね。

```
E000 3A 10 E0 E6 F0 32 11 E0 76 FF 00 FF 00 FF 00 FF
E010 FF F0
```

実行前と比べてみてください。$E011番地が変化していますね。

$FFの下位4ビットがマスクされて$F0となり$E011番地に格納されたことに注意してください。

3・4　8ビットの足し算

　機械語レベルで足し算をするとどうなるでしょうか。ひとつ考えてみましょう。ここでＰＣ－８８０１のモニタプログラムがインテル形式のアセンブラを採用しているので、インテル形式でやってみましょう。本当はザイログ形式を使うと非常に簡単になるのですが、ここではインテル形式でやるのも教育的見地から面白いと思います。

E000　機械語プログラム
E008　未使用
E010　足される数
E011　足す数
E012　和
　　　未使用
E5FF

メモリマップ

　はじめにメモリの使い方を次のように約束しておきましょう。

　　＄Ｅ０１０番地――足される数
　　＄Ｅ０１１番地――足す数
　　＄Ｅ０１２番地――和

　あらかじめ、こういうふうに考えておいた方が楽になります。

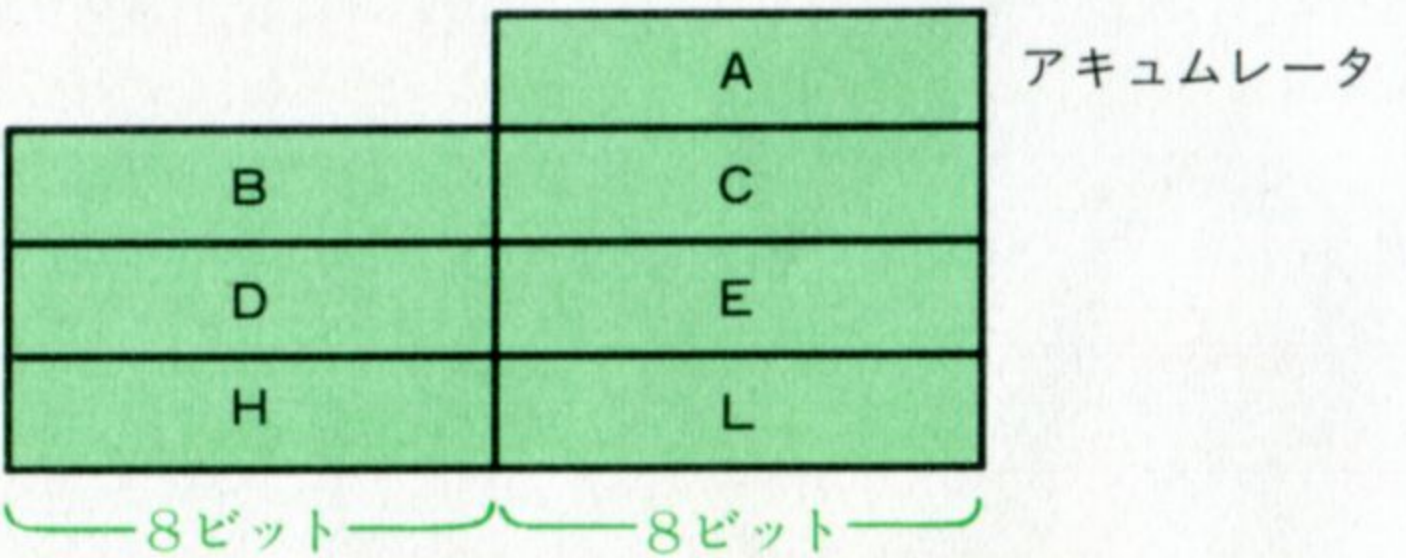
　われわれの使用したマイクロプロセッサのレジスタはアキュムレータだけでしたが、本プログラムから少し別のものを使うことにしましょう。

　ＰＣ－８８０１のマイクロプロセッサ Z80 にはたくさんのレジスタがあります。先のＺ80の説明をしたときにＺ80はちょうど８０８０を２つあわせたような形をしているといいました。Ｚ80の場合も基本的には８０８０と同じです。基本的なレジスタは、

　Ａ，Ｂ，Ｃ，Ｄ，Ｅ，Ｈ，Ｌ　と　ＩＸ，ＩＹ

です。ほかにもあるのですが、でてきたときに勉強することにしましょう。

　Ａはアキュムレータで、これはすでに出てきました、Ｂ，Ｃ，Ｄ，Ｅ，Ｈ，Ｌがはじめてでてきたレジスタです。ＢとＣ，ＤとＥ，ＨとＬはペアにして使います。下の図をごらんください。

	A	アキュムレータ
B	C	
D	E	
H	L	
8ビット	8ビット	

　使い方に関しては、プログラムする人の自由だといえば自由なのですが、ある程度のきまりもあります。とくにＨとＬのレジスタのペア（これをＨＬレジスタといいます）については約束があります。

ＨＬレジスタ
データのポインタ

　ＨＬレジスタは**データのポインタ**として使うのです。

データのポインタというのは『指し示すもの』という位の意味で、データの格納番地を示します。Hレジスタにはデータの格納番地の上位ビット (Higher bits) をLレジスタにはデータの格納番地の下位ビット (Lower bits) をしまうことになります。

ポインタとしてのHLレジスタの役目は、データが必要となったときに、どこにデータがあるかを指し示してくれるのです。どうしてこんなものが必要となるのか疑問に思われる人も多いと思います。その理由は、データがメモリの中で連続した形で並んでいるときには、ポインタを使うと機械語プログラムが短くてすみ、演算速度も早くなるからです。ただし、アセンブリ言語で書いた場合には短くなっていることはわかりにくいと思います。

もう1つ注意しておくことがあります。それは**レジスタM**の存在です。このレジスタは物理的には存在しないのですが、HLレジスタで指し示されたメモリ番地の中味をレジスタMと呼んでいるのです。

レジスタM

だから、

MOV　A, M

とあれば、HLレジスタで指し示された番地（レジスタM）の中味をアキュムレータAに移動することで、たとえば次のようならば、

MOV
(MOVE FROM REGISTER TO REGISTER)

MOV　M, A

アキュムレータAの中味をHLレジスタで指し示された番地（レジスタM）に移動することになります。

これだけの準備がありますと、足し算は少し気どったやり方で実行できます。アセンブリ言語のプログラムをまずごらんになってください。

```
E000    21 E010      LXI  H,E010   ← 2バイトの数値データをHLレジスタへ代入
E003    7E           MOV  A,M      ← HLレジスタ対で指定された番地の中味(レジスタM)をアキュムレータへ移動
E004    23           INX  H        ← HLレジスタ対の値を1だけ増やす
E005    86           ADD  M        ← レジスタMの中味をアキュムレータの中味に加える
E006    23           INX  H        ← HLレジスタ対の値を1プラス
E007    77           MOV  M,A      ← アキュムレータの中味をレジスタ対HLで指定された番地にしまう
E008    76           HLT           ← 終わり
```

LXIは16ビットのデータをレジスタ対に直ちに代入するものです。Xはダブルの対を表します。ついでにいうと、LはLOADの略で、IはIMMEDIATEの略です。

LXI
(LOAD A 16-BIT VALUE, IMMEDIATE, INTO A REGISTER PAIR)

INXはレジスタ対の値を1だけ増加（INCREMENT）させるものです。

INX
(INCREMENT REGISTER PAIR)

それではプログラムの内容を説明しましょう。

まずHLレジスタにE010を代入します。IMMEDIATEというのは$E010という値を代入するので、$E010番地の内容を代入するのではないことに注意してください。

次に　MOV A, M　があります。これはHLレジスタ対によって指定された番地$E010の中味をアキュムレータAに移動させています。たとえば$E010番地に1が入っていれば、アキュムレータAに1が移動したことになります。

INX H　は、HLレジスタ対の値を1だけ増やす、つまり、$E010を$E011へと増加します。

ADD
(ADD REGISTER OR MEMORY TO ACCUMULATOR)

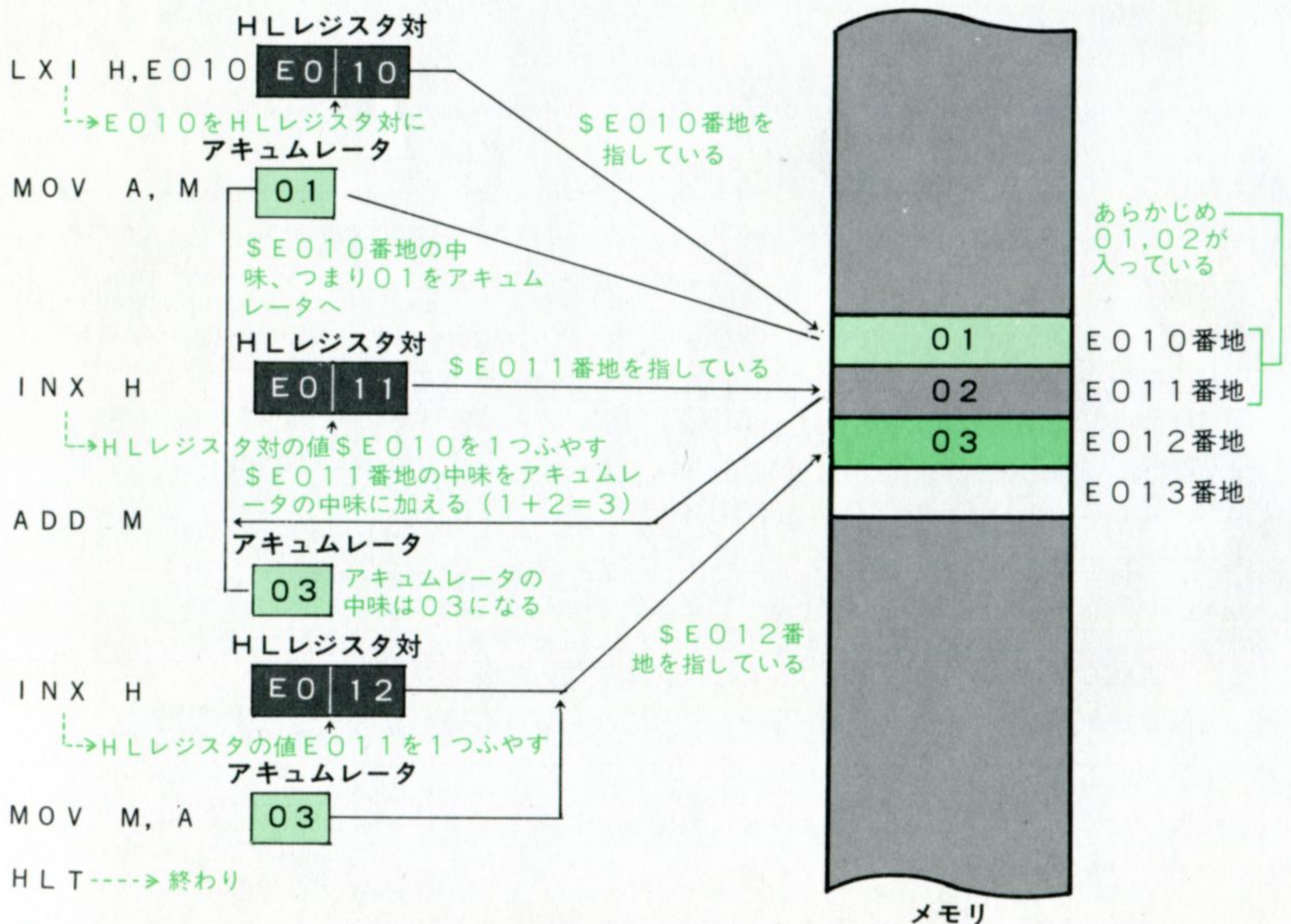
ADD M　は、HLレジスタ対によって指定された番地、$E011番地の中味をアキュムレータAの中味に加えるものです。いま$E011番地に2が入っているとしますと、アキュムレータの中味は1+2=3になりました。

INX H　は、HLレジスタ対の値をさらに1だけ増やして、$E012へと進めます。

MOV M, A　は、アキュムレータAの中味をHLレジスタ対によって指定された番地$E012にしまいなさいということです。

HLTは、これで終わりということです。

どうですか？　わかりましたか。

HLレジスタ対
LXI H,E010　E0 | 10
E010をHLレジスタ対に
$E010番地を指している
アキュムレータ
MOV A, M　01
$E010番地の中味、つまり01をアキュムレータへ
あらかじめ01, 02が入っている
HLレジスタ対
INX H　E0 | 11
$E011番地を指している
HLレジスタ対の値$E010を1つふやす
$E011番地の中味をアキュムレータの中味に加える（1+2=3）
ADD M
アキュムレータ
03　アキュムレータの中味は03になる
HLレジスタ対
INX H　E0 | 12
$E012番地を指している
HLレジスタの値E011を1つふやす
アキュムレータ
MOV M, A　03
HLT----→終わり
01　E010番地
02　E011番地
03　E012番地
E013番地
メモリ

実行前のメモリの状態は下のようになっています。Dコマンドで調べてみてください。

```
E000 21 10 E0 7E 23 86 23 77 76 FF 00 FF 00 FF 00 FF
E010 01 02 00
     ↑  ↑
     │  └──────────足す数
     └─────────────足される数
```

プログラム実行後のメモリの状態は次のようです。

０１＋０２＝０３

が＄Ｅ０１２番地に入っているでしょう。

```
E000 21 10 E0 7E 23 86 23 77 76 FF 00 FF 00 FF 00 FF
E010 01 02 03
           ↑
           └──────答
```

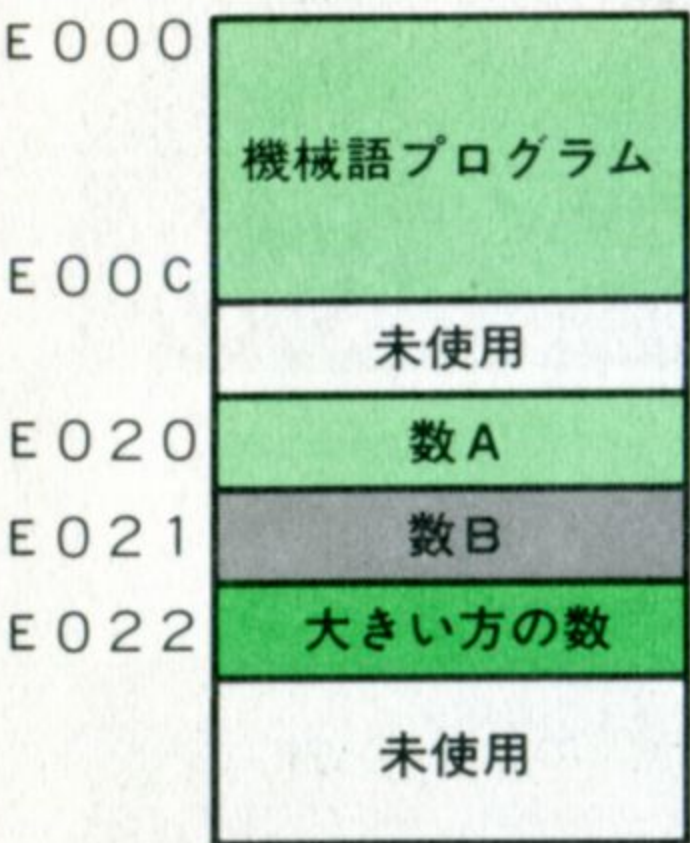
アドレッシングを勉強する必要性

本書はインテル系のアセンブラを使っているので、インテル８０８０系のアドレッシング（番地指定）を主に使っています。言葉が堅くなるのですが、ＨＬレジスターをポインタとして使う時などは implied memory addressing（暗目のメモリ番地指定）といい、direct memory addressing（直接のメモリ番地指定)というものもあります。この他に immediate memory reference（即値メモリ参照）というのもあります。ところがＰＣ-８８０１の本体であるマイクロプロセッサＺ８０には大変豊富なアドレッシングの方法が準備されており、これを使わない手はありません。列挙してみると次のようになります。

インテル８０８０にあるもの
- ⊙ Implied
- ● implied block transfer
- ● implied stack
- ● indexed
- ⊙ direct
- ● program relative
- ● base page
- ● register indirect
- ⊙ immediate

まだまだ勉強は奥深いものだということがわかります。

3・5 大きい数をさがすこと

番地	内容
E000	機械語プログラム
E00C	未使用
E020	数A
E021	数B
E022	大きい方の数
	未使用

メモリマップ

２つの数ＡとＢがあるとすると、どちらが大きいかを調べるには、ふつう引き算をして比較することになります。いろいろなやり方が考えられるのですが、ここでは前の節で勉強したＨＬレジスタを使うやり方を採用することにします。

ここでメモリの使い方を左のように決めておきます。

＄Ｅ０２０番地——比較される数Ａ
＄Ｅ０２１番地——比較される数Ｂ
＄Ｅ０２２番地——大きい方の数

まずプログラムをごらんください。

```
E000    21 E020      LXI   H,E020
E003    7E           MOV   A,M
E004    23           INX   H
E005    BE           CMP   M
E006    D2 E00A      JNC   E00A
E009    7E           MOV   A,M
E00A    23           INX   H
E00B    77           MOV   M,A
E00C    76           HLT
```

- LXI H,E020 ← ＨＬレジスタに＄Ｅ０２０を転送
- MOV A,M ← レジスタＭの中味をアキュムレータに入れる
- INX H ← ＨＬレジスタを１つふやす
- CMP M ← レジスタＭとアキュムレータの内容を比較
- JNC E00A ← キャリフラグが０のとき＄Ｅ００Ａ番地へ飛ぶ
- MOV A,M ← レジスタＭの中味をアキュムレータへ移動する
- INX H ← ＨＬレジスタの中味を１つふやす
- MOV M,A ← アキュムレータの中味をレジスタＭへ
- HLT ← 終わり

最初にＨＬレジスタに＄Ｅ０２０を入れています。ＨＬレジスタはデータのポインタに使うのでした。従って２行目の MOV A, Mによって、アキュムレータＡにはＨＬレジスタの指している番地＄Ｅ０２０番地の内容が入ります。Ｍは一種の仮想的なレジスタとして使われているのです。

アキュムレータＡに＄Ｅ０２０番地の内容、すなわち比較される数Ａを代入しますと、３行目の　ＩＮＸ Ｈ　でＨＬレジスタの内容を１だけふやします。ＩＮＸはINCREMENT REGISTER PAIRでした。従ってＨＬレジスタは＄Ｅ０２１番地を指すことになります。

CMP
(COMPARE SOURCE DATA)

第４行目の　ＣＭＰ Ｍ　は、レジスタMすなわちＨＬレジスタの指し示しているメモリ番地の内容をアキュムレータＡの内容と比較します。実際には、次の操作が行なわれています。

(アスキュレータの内容) − (レジスタMの内容)

ＣＭＰ　Ｍ　は引き算とは違います。引き算は　ＳＵＢ　Ｍ　という命令が準備されています。ＣＭＰ　Ｍ　の場合は引き算した結果、2つの数のどちらが大きいかだけに興味があって、引き算の答そのものには興味がないのです。それでは大小の判定はどうしたらできるのでしょうか？

フラグというものを使えばよいのです。

符号（Ｓｉｇｎ）
ゼロ（Ｚｅｒｏ）
補助キャリ
パリティ
キャリ(Ｃａｒｒｙ)

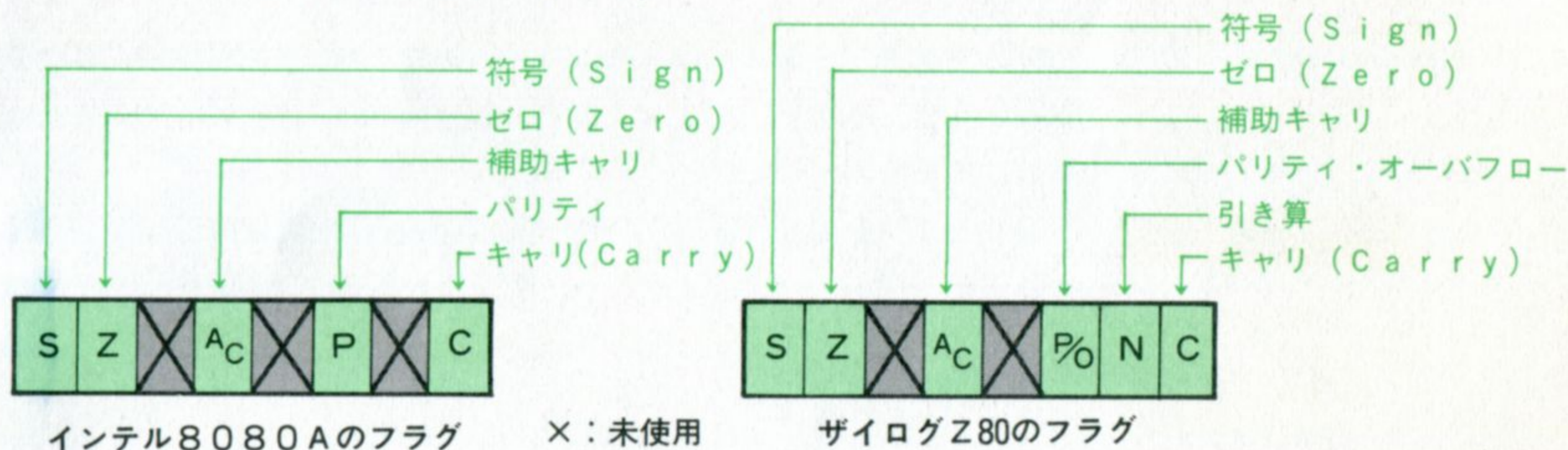
S	Z	×	AC	×	P	×	C

インテル８０８０Ａのフラグ

×：未使用

符号（Ｓｉｇｎ）
ゼロ（Ｚｅｒｏ）
補助キャリ
パリティ・オーバフロー
引き算
キャリ（Ｃａｒｒｙ）

S	Z	×	AC	×	P/O	N	C

ザイログＺ80のフラグ

ここで８０８０の興亡史について一言ふれておかねばなりません。マイクロプロセッサ・インテル８０８０は日本人の嶋正利氏によって設計されました。ところが同じ嶋正利氏が、ザイログ社に移って設計したのがザイログＺ８０です。ですから2つのマイクロプロセッサはよく似ています。しかし、Ｚ８０は８０８０の改良型ですから機能もぐんと増加しています。Ｚ８０は圧倒的なシェアを持っており、わがＰＣ-８８０１もＺ８０を採用しています。ところがややこしいことにＺ８０はハードウェア的にはすぐれていたものの、ソフトウェアの供給が追いつきませんでした。むしろインテル社がソフトウェアに力を入れたために、Ｚ８０のユーザは互換性のあるインテル社系のソフトウェアを採用してしまいました。これが奇妙な事態をひき起すことになってしまったのです。

つまりハードウェアはＺ８０を採用しながら、ソフトウェアはザイログのアセンブリ言語を採用することなく、インテル社のアセンブリ言語を採用するという事態が生じてしまいました。これはＣＰ／Ｍというオペレーティング・システムが支配的なシェアを持ったために確定的になってしまいました。本書がインテルのアセンブリ言語を採用しているのもそのためです。技術は何とも曲折した道を歩むものです。

インテル社のアセンブリ言語を採用したのではＺ８０の性能を完全に引き出すことはできないのですが、総合的に判断しますと、インテル社のアセンブリ言語を使った方が有利な部分が多いと私は判断しています。しかし、あるべきフラグを使わなかったり、何とも気味の悪い部分もないわけではありません。この辺で歴史的事情をよくご理解ください。

MSB
(MOST SIGNIFICANT BIT)

基本的なフラグについて説明しておきましょう。

符号フラグ

符号（Ｓｉｇｎ）フラグというのは、算術演算や代数演算を実行した結果の最上位ビット(ＭＳＢ)を移します。計算結果の正負を示すことになります。

ゼロフラグ

ゼロ（Ｚｅｒｏ）フラグは、算術演算や代数演算を実行した結果が０であるかどうかを示します。０のときにはゼロフラグが１になります。

補助キャリフラグ

補助キャリ（Ａｕｘｉｌｉａｒｙ Ｃａｒｒｙ)フラグは第３ビットから第４ビットの桁上げを示します。これはＳＣＤ演算というものを簡単にするものです。

パリティフラグ

パリティ（Ｐａｒｉｔｙ）フラグは算術演算や代数演算の結果、１の数が偶数個のとき１となり、１の数が奇数個のときは０となります。

キャリフラグ

キャリ（Ｃａｒｒｙ）フラグは算術演算の結果、最上位ビットからの桁上がりを示します。

●キャリの動作は後の節で詳しく説明します。

いま問題としたいのはキャリフラグです。キャリフラグは次のように値を変えます。

●A＞Bは、AはBより小さい。

●A≧Bは、AはBより大きいか等しい。（小さくない。）

●（アキュムレータの内容）＜（レジスタMの内容）のとき

キャリフラグ＝1

●（アキュムレータの内容）≧（レジスタMの内容）のとき

キャリフラグ＝0

JNC
(JUMP IF NO CARRY)

第５行目のＪＮＣは JUMP IF NO CARRY で、桁上げがないとき、すなわちキャリフラグ＝０のときには指定された番地へとびます。このプログラム例では＄Ｅ００Ａ番地です。本当はラベル付けを許してくれると楽なのですが、ＰＣ-８８０１のアセンブラはそういう仕組になっていません。

つまり、アキュムレータの内容の方がレジスタMの内容より小さくないとき、言いかえると、＄Ｅ０２０番地の数の方が＄Ｅ０２１番地にしまわれている数より小さくないときには、プログラムの処理は＄Ｅ００Ａ番地へとぶのです。

＄Ｅ００Ａ番地からのプログラムをみてみますと、ＩＮＸ Ｈ で、ＨＬレジスタの中味を１番地すすめて＄Ｅ０２２番地にしています。さらに ＭＯＶ Ｍ，Ａ でアキュムレータの内容をＨＬレジスタが指し指している番地＄Ｅ０２２番地にしまっています。

次にＪＮＣでキャリが１だった場合、すなわちアキュムレータの内容の方がレジスタMの内容より小さいとき、言いかえると＄Ｅ０２０番地の数の方が＄Ｅ０２１番地の数より小さい場合には、ジャンプの

条件をみたさないので、６行目の MOV　A，M の処理になります。このとき、HLレジスタは＄E０２１番地を指し示しているので、アキュムレータには、大きい方の数をしまうことになります。

７行目でHLレジスタの中味を１番地すすめ＄E０２２とし、８行目の MOV M，A で、HLレジスタの指し示している＄E０２２番地にアキュムレータの中味、つまり大きい方の数をしまうことになります。

結局どちらの経路をたどっても＄E０２２番地には大きい方の数がしまわれることになるのです。

最後に一言、『大きい』ということは記号的には>で、『小さくない』ということ≧とは、厳密にいうと違います。この場合には等号を入れても最終的な結果にひびかなかったのですが、プログラムによっては差がでてしまうことがあります。しかもそれが決定的に効いてくることがあるので、これから先、細心の注意を持ってプログラムをしなければなりません。

面倒と思わないでください。パズルと思えばけっこう楽しいものです。

プログラム実行前のメモリの状態は下のようになっています。

```
E000 21 20 E0 7E 23 BE D2 0A E0 7E 23 77 76 78 E6 0F
E010 23 77 76 00 10 19 24 31 00 FF 00 FF 00 FF 00 FF
E020 01 02 00
     └──┘
       ↑
       └────────比較される数
```

プログラム実行後のメモリの状態は次のようです。

０１と０２のうち大きい方の０２が＄E０２２番地に入っていることがわかります。正しく実行されたのです。

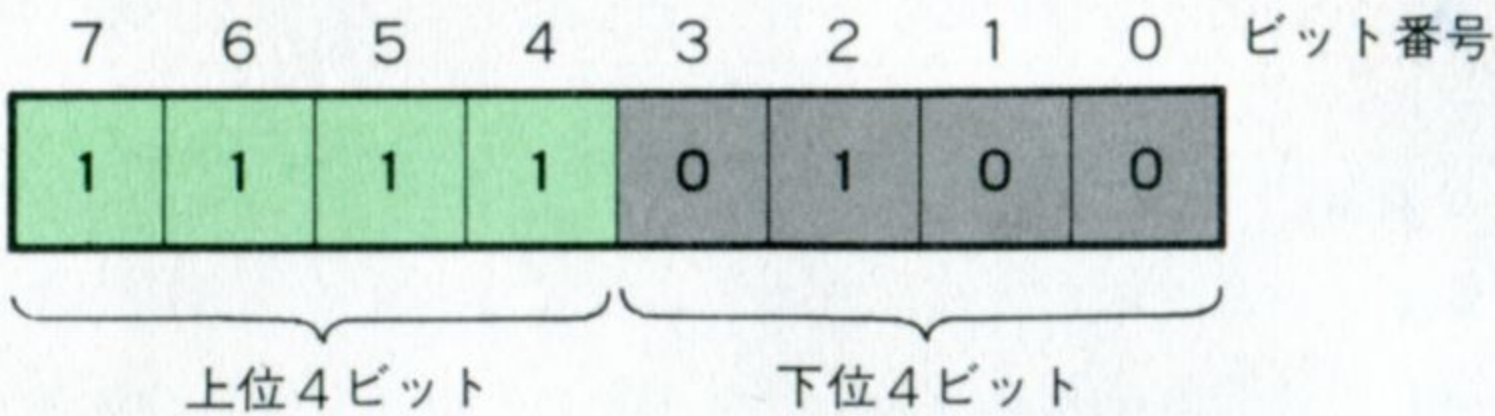
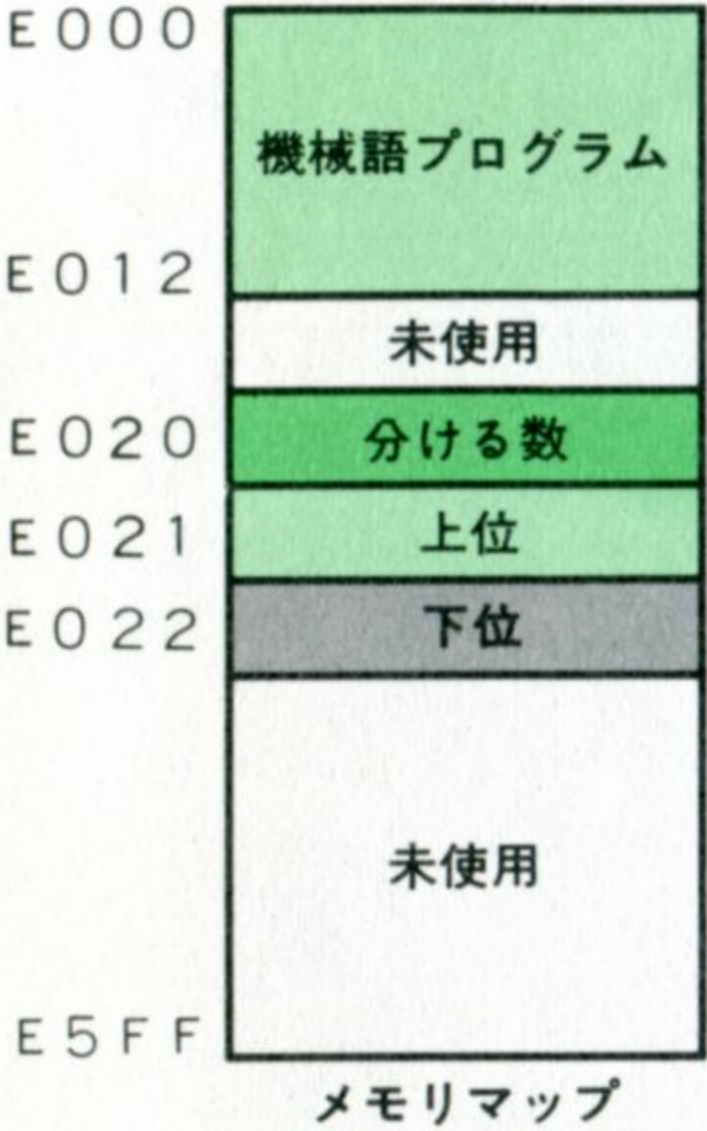
```
E000 21 20 E0 7E 23 BE D2 0A E0 7E 23 77 76 78 E6 0F
E010 23 77 76 00 10 19 24 31 00 FF 00 FF 00 FF 00 FF
E020 01 02 02
           ↑
           └────────大きいほうがSE020番地に入っている
```

3・6 メモリの内容を2つに分ける

大変わかりにくい表題なのですが、PC-8801のマイクロプロセッサであるZ80は8ビット・マイクロプロセッサなので、扱うデータは8ビット単位です。このことは上位4ビット、下位4ビットのそれぞれで0～15までの数を表現できることを意味します。16進法でいえば0～Fまでの数を表現できるのです。

アドレス	内容
E000	機械語プログラム
E012	未使用
E020	分ける数
E021	上位
E022	下位
	未使用
E5FF	

メモリマップ

7	6	5	4	3	2	1	0	ビット番号
1	1	1	1	0	1	0	0	
上位4ビット				下位4ビット				

だから、あるメモリ番地の内容がF4であることもあります。表題の意味は0Fと04にわけて別々のメモリ番地に格納しようということです。

例によってメモリの使い方を決めておきましょう。

プログラムを下に示します。いままで勉強してきた命令ばかりですから、よくわかると思います。

```
E000    21 E020      LXI  H,E020  ← $E020をHLレジスタに
E003    7E           MOV  A,M     ← レジスタMの中味をアキュムレータに
E004    47           MOV  B,A     ← アキュムレータの中味をレジスタBにうつす
E005    0F           RRC          ← アキュムレータの各ビットが1ビット右へ
E006    0F           RRC            ずれる。第7ビットには何が入るかわから
E007    0F           RRC            ない。これを4回くり返すから、上位4ビ
E008    0F           RRC            ットがそのまま下位4ビットへ移行する。
E009    E6 0F        ANI  0F      ← マスクをかける
E00B    23           INX  H       ← HLレジスタを1ふやす
E00C    77           MOV  M,A     ← アキュムレータの中味をレジスタMへ
E00D    78           MOV  A,B     ← レジスタBの中味をアキュムレータへ
E00E    E6 0F        ANI  0F      ← マスクをかける
E010    23           INX  H       ← HLレジスタを1ふやす
E011    77           MOV  M,A     ← マスクをされたアキュムレータの中
                                    味をレジスタM、つまり$E022
                                    番地へ格納する
E012    76           HLT          ← 終わり
```

プログラムを見ていきましょう。まずHLレジスタに$E020をロードします。次に　MOV A, M　でアキュムレータAにHLレジスタの指し示している$E020番地の中味を代入します。さらに、

ＭＯＶ Ｂ，Ａでレジスタ Ｂにアキュムレータ Ａの中味をうつします。

結局、＄Ｅ０２０番地の中味がレジスタＢにロードされることになります。つまり、最初のデータをレジスタＢに待避させておきます。

次にＲＲＣを４回くり返して、アキュムレータの上位４ビットを下位４ビットにうつします。上位４ビットには何が入るかわかりません。

RRC
(ROTATE ACCUMULATOR RIGHT INTO CARRY)

上位	下位

アキュムレータＡ

ＲＲＣを４回くり返す →

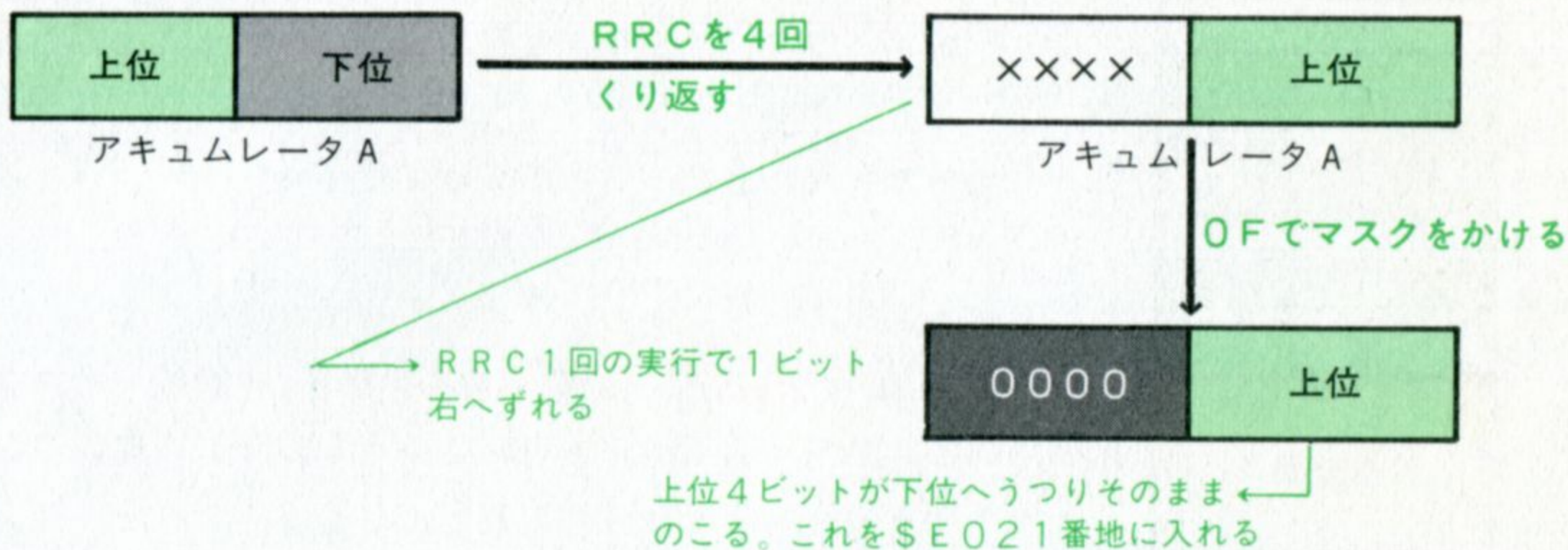
××××	上位

アキュムレータＡ

→ ＲＲＣ１回の実行で１ビット右へずれる

↓ ０Ｆでマスクをかける

００００	上位

上位４ビットが下位へうつりそのままのこる。これを＄Ｅ０２１番地に入れる

そこで０Ｆでマスクをかけます。ＡＮＩ ＯＦ でマスクします。次に ＩＮＸ Ｈ でHLレジスタのデータポインタを１すすめて、＄Ｅ０２１とします。ＭＯＶ Ｍ，Ａ でアキュムレータの中味を＄Ｅ０２１番地に格納します。

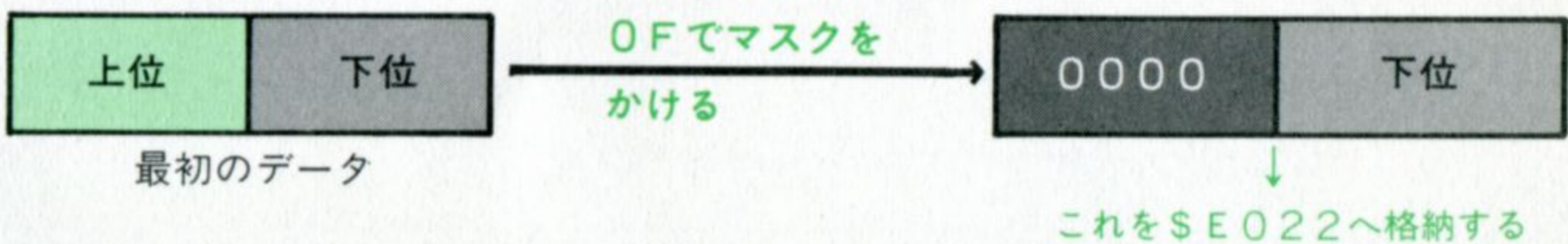
さらに ＭＯＶ Ａ，Ｂ で＄Ｅ０２０番地の内容をレジスタＢに待避させていたものを再びアキュムレータＡにうつします。さらに、ＡＮＩ ＯＦ でマスクをかけてアキュムレータの下位だけを残します。

上位	下位

最初のデータ

０Ｆでマスクをかける →

００００	下位

↓ これを＄Ｅ０２２へ格納する

そして ＩＮＸ Ｈ でHLレジスタのデータポインタを１すすめて＄Ｅ０２２とします。ＭＯＶ Ｍ，Ａ でアキュムレータの中味を＄Ｅ０２２番地に格納します。

ここで、実際にプログラムを実行してみてください。

まず、Ｓコマンドで＄Ｅ０２０番地に＄Ｆ４を、＄Ｅ０２１番地に＄００を、＄Ｅ０２２番地に＄００を入れます。次に、ｈ］ＧＥ０００⏎として、ダンプリストをとってみてください。

結果は次のようになります。

```
E000 21 20 E0 7E 47 0F 0F 0F 0F E6 0F 23 77 78 E6 0F
E010 23 77 76 FF 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF
E020 F4 0F 04
```

（0F 04）← Ｆ４が２つに分かれている

3・7　16ビットの足し算

アドレス	内容	
E000	機械語プログラム	
E00B		
	未使用	
E010	被加数	下位
E011		上位
E012	加　数	下位
E013		上位
E014	和	下位
E015		上位
	未使用	
E5FF		

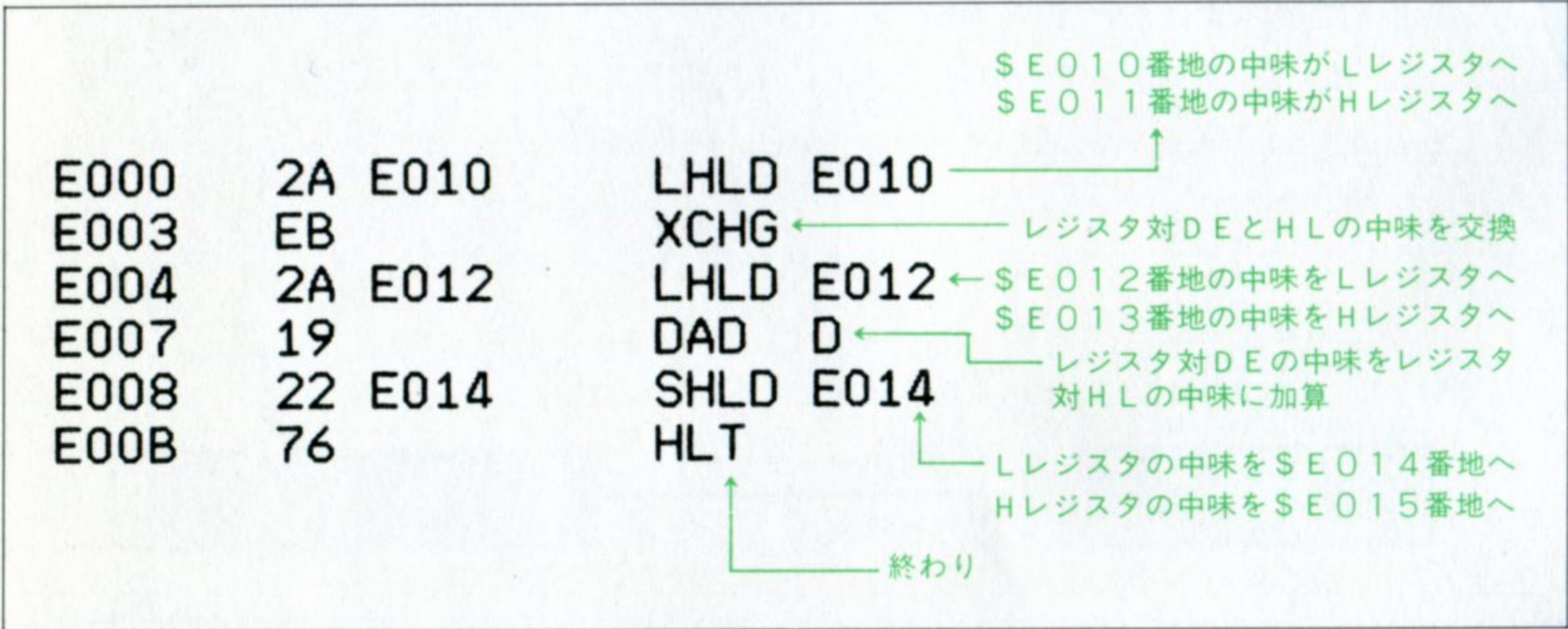
メモリマップ

さて８ビットでは０から$2^8-1=255$までの数字しか表現できませんでした。符号つきともなると、−128から＋127までの数字しか表現できないのです。これではとても実用にはなりません。そこで８ビットのマイクロプロセッサでも、８ビットのレジスタを２つつなげて16ビットのデータを扱います。こうすると、０から、

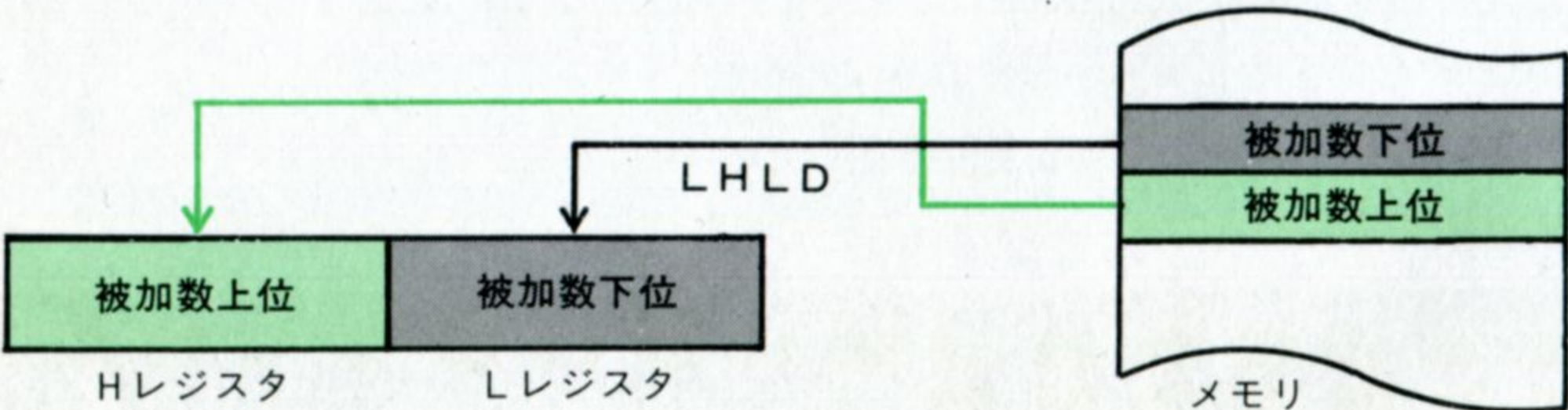
$$2^{16}-1=65535$$

までの数字を表現できることになり、少しマシになります。もっとも０から65535といっても整数に限られています。一般のパソコンが実数を扱えるのは浮動小数点パッケージというものを持っているからです。

まずメモリの使用状況を決めておきましょう。これをキチンとしておかないといけないのです。プログラムは下図に示します。

```
E000    2A E010        LHLD E010
E003    EB             XCHG
E004    2A E012        LHLD E012
E007    19             DAD  D
E008    22 E014        SHLD E014
E00B    76             HLT
```


LHLD
(LOAD H AND L
REGISTERS DIRECT)

最初にＬＨＬＤについて説明しましょう。この命令はＨＬレジスタにメモリのある場所から連続して２バイトロードするものです。ＨＬレジスタにどういう順に入るかが問題ですが、メモリの先行する１バイトがＬレジスタに、次の１バイトはＨレジスタに代入されることになります。

メモリの＄Ｅ０１０番地から＄Ｅ０１３番地へどうして被加数と加数を下位、上位の順に格納しなければならないかがわかるでしょう。

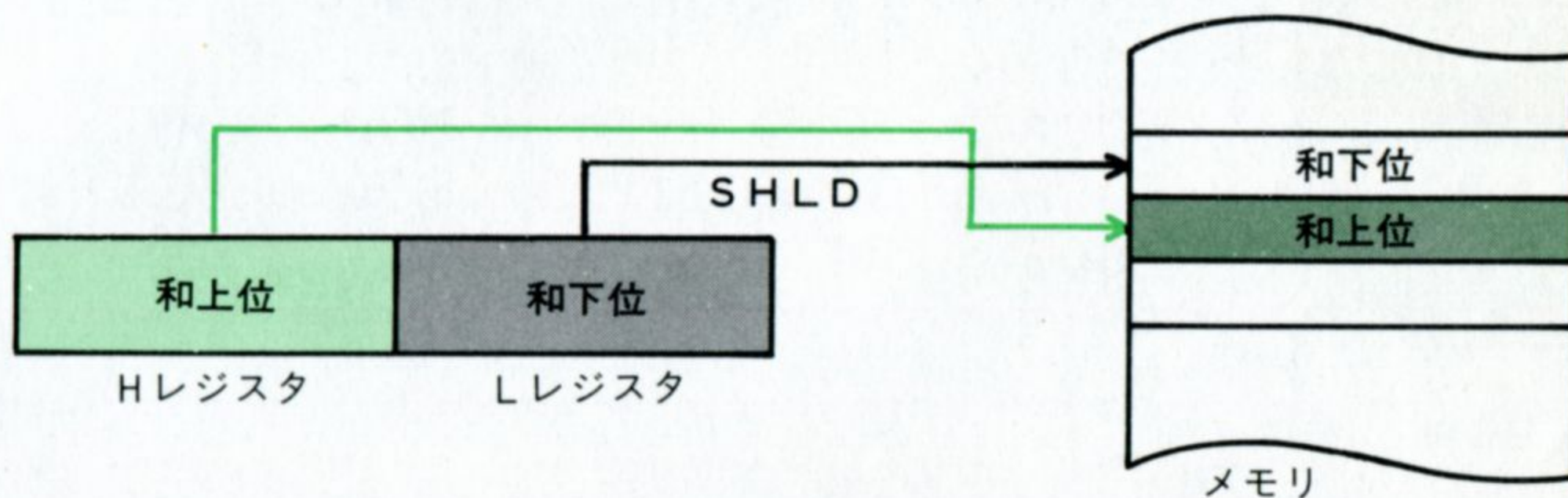
ＸＣＧはＤＥレジスタとＨＬレジスタの中味を交換します。これで被加数はＤＥレジスタに格納されることになります。次に、ＬＨＬＤ　Ｅ０１２　で加数をＨＬレジスタにロードします。

XCHG
(EXCHANGE DE AND HL REGISTERS' CONTENTS)

ＤＡＤ　Ｄ　は、ＤＥレジスタ対の中味をＨＬレジスタ対の中味に加算します。つまり16ビットの加算をしてくれることになります。

DAD
(ADD A REGISTER PAIR TO H AND L)

最後にＳＨＬＤについて説明しましょう。この命令はＨＬレジスタの中味をメモリのある場所から連続して２バイトに格納するものです。Ｌレジスタの中味を先行するメモリ番地へ、Ｈレジスタの中味を続くメモリ番地へ格納することになります。

SHLD
(STORE H AND L REGISTERS DIRECT)

ここでＬＸＩ，ＬＨＬＤの違いについて注意しておきましょう。両者とも16ビットのデータをロードするのですが、ＬＸＩが即値（ＩＭＭＥＤＩＡＴＥ），ＬＨＬＤが（ＤＩＲＥＣＴ）であるという違いがあります。どう違うか考えてみることにしはしょう。右図のような設定のとき、次の命令でＨＬレジスタにはどんな値が代入されることになるでしょうか。

❶　ＬＸＩ　Ｈ，Ｅ０２０

❷　ＬＨＬＤ　Ｅ０２０

❶の場合、ＨレジスタにはＥ０，Ｌレジスタには２０が、❷の場合Ｈレジスタには０７，Ｌレジスタには２０が代入されることになります。少し考えてみてください。何ともまぎらわしいですね。

ここで、被加数を＄０００１、加数を＄０００２として実際に実行させてみましょう。Ｓコマンドでデータ入力をしますが、この場合下位、上位の順に入りますから、＄Ｅ０１０番地に０１を、＄Ｅ０１１番地に００を、＄Ｅ０１２、＄Ｅ０１３番地にそれぞれ０２、０１をセットします。結果が入る＄Ｅ０１４、＄Ｅ０１５番地は００にしておきます。結果は次のようになります。

```
E000 2A 10 E0 EB 2A 12 E0 19 22 14 E0 76 76 78 E6 0F
E010 01 00 02 00 03 00
```

3·8 表をひいて計算すること

マイクロコンピュータの応用においては表を使うことがよくあります。２乗計算をするプログラムをご覧ください。

E000	機械語プログラム
E00E	未使用
E010	問題
E011	答
	未使用
E020	表
E027	
	未使用
E800	

メモリマップ

```
E000   3A E010      LDA  E010
E003   6F           MOV  L,A
E004   26 00        MVI  H,00
E006   11 E020      LXI  D,E020
E009   19           DAD  D
E00A   7E           MOV  A,M
E00B   32 E011      STA  E011
E00E   76           HLT
```

まずあらかじめ、Ｓコマンドを使って＄Ｅ０２０番地から＄Ｅ０２７番地まで、$0^2=0$，$1^2=4$，$3^2=9$，$4^2=10$，$5^2=19$、$6^2=24$，$7^2=31$を入れておきます。つまり、２乗の表を入れておくわけです。

●10進法では、$4^2=16$ですが、16進法では$4^2=10$です。同様に、$5^2=19$、$6^2=24$、$7^2=31$となります。

第１行目で＄Ｅ０１０番地の値をアキュムレータに代入します。

第２行目の　ＭＯＶ　Ｌ，Ａ　でアキュムレータＡの中味をＬレジスタに代入します。

第３行目の　ＭＶＩ　Ｈ，００　でHレジスタに0を代入します。

第4行目の　ＬＸＩ　Ｄ，Ｅ020でDEレジスタにＥ０２０という値を代入します。これは表のベースアドレスを与えるものです。

第５行目の　ＤＡＤ　DでＨＬレジスタとＤＥレジスタの値の和が作られます。ベースアドレスと与えられた数値の和を作り、再びＨＬレジスタにしまいます。

ＭＶＩ
(MOVE IMMEDIATE DATA TO REGISTER)

第６行目の　ＭＯＶ　Ａ，ＭでＨＬレジスタの指し示す番地のデータがアキュムレータに入ることになります。

第７行目の　ＳＴＡ　Ｅ０１１　で、答はＥ０１１番地に代入されて終わりとなります。

プログラム実行前のメモリの状態を下に示しておきます。

```
E000 3A 10 E0 6F 26 00 11 20 E0 19 7E 32 11 E0 76 0F
E010 03 00 00 00 00 00 24 31 00 FF 00 FF 00 FF 00 FF
E020 00 01 04 09 10 19 24 31
```

表が入っている

それでは、プログラム実行後のメモリの状態を調べてみてください。次のようになります。

```
E000 3A 10 E0 6F 26 00 11 20 E0 19 7E 32 11 E0 76 0F
E010 03 09 00 00 00 00 24 31 00 FF 00 FF 00 FF 00 FF
E020 00 01 04 09 10 19 24 31
```

$E011番地に答が入っている

＄Ｅ０１０番地にしまわれた０３の２乗の答９が＄Ｅ０１１番地に入っています。

表を多用するのは三角関数や超越関数の計算です。高速Ｆｏｕｒｉｅｒ（フーリエ）変換とよばれる手法などで用いられます。ロボットや自動車のスピード調整用にも使われますし、符号の変換用にも用いられます。

三角関数の表を利用すること

マイクロコンピュータでは、三角関数をどのように作っているのでしょうか。一番有名な作り方はマクローリン展開を利用するもので次式を利用します。

$$\sin X = X - \frac{X^3}{3!} + \frac{X^5}{5!} - \frac{X^7}{7!} + \frac{X^9}{9!} - \cdots\cdots$$

無限個の項をとれるわけではないので、どの位項数をとれるかによって関数の精度が決まります。ふつうマイクロソフト系のＢＡＳＩＣでは単精度の場合５項までとるといわれています。誤差を考えると５ケタ位までｓｉｎ関数の値としては正しいことになります。演算時間は１ｍｓ（ミリセコンド）程度になります。これでは実用的に遅すぎるという場合もあります。その時には表を引く方法を使います。ｓｉｎ関数の値は次のように表現できます。

$$\sin X = \boxed{符号}.\ D_6\ D_5\ D_4\ D_3\ D_2\ D_1\ D_0$$

たとえば　$X=0°$ のとき　$D_6D_5D_4D_3D_2D_1D_0$ の値は０００００００
　　　　　$X=10°$ のとき　$D_6D_5D_4D_3D_2D_1D_0$ の値は００１０１１０
　　　　　$X=20°$ のとき　$D_6D_5D_4D_3D_2D_1D_0$ の値は０１０１０１１
　　　　　$X=30°$ のとき　$D_6D_5D_4D_3D_2D_1D_0$ の値は１００００００

などとして表を作っておくと、演算時間は１ケタ程度早くなるといわれています。

3・9　16ビットの1の補数

だいぶ勉強が進んできたので、表題を見ただけで、あれはこうするんだろうか、こうしたら面白いなと読者の皆さんも自分で考えられるようになっているでしょう。16ビットの1の補数も８ビットずつの補数をとればよいということもおわかりと思います。

E000 機械語プログラム
E00C
未使用
E010 下位
E011 上位
E012 答 下位
E013 答 上位
未使用

メモリマップ

```
E000   2A E010    LHLD E010
E003   7D         MOV  A,L
E004   2F         CMA
E005   6F         MOV  L,A
E006   7C         MOV  A,H
E007   2F         CMA
E008   67         MOV  H,A
E009   22 E012    SHLD E012
E00C   76         HLT
```

プログラムを見ていきましょう。

第１行目では＄Ｅ０１０番地の中味をＬレジスタに、＄Ｅ０１１番地の中味をＨレジスタにそれぞれとり込んでいます。

第２行目では、Ｌレジスタの中味をアキュムレータにうつし、第３行目でその補数を作っています。つまり、Ｌレジスタのデータが＄ＦＦだったとすると、アキュムレータの中味は＄００に変わります。

〈ＥＸ．$(\mathrm{FF})_6$は$(1111\ \ 1111)_2$、この補数は$(00000000)_2$〉

この補数を第４行目でもう一度Ｌレジスタにもどします。

第５行目から第７行目では同じことをＨレジスタについて、つまり上位ビットのデータの補数を作り、再びＨレジスタに格納しています。

第８行目では、Ｌレジスタの中味を＄Ｅ０１２番地に、Ｈレジスタの中味を＄Ｅ０１３番地に格納します。

これで16ビットの補数計算は終了しているのです。簡単でしょう。

それでは、＄Ｅ０１０、＄Ｅ０１１番地にそれぞれＦＦをセットしておき、プログラムを実行してみましょう。実行後のメモリの状態を下に示します。

```
E000 2A 10 E0 7D 2F 6F 7C 2F 67 22 12 E0 76 FF 00 FF
E010 FF FF 00 00
```

＄Ｅ０１０，＄Ｅ０１１番地のＦＦＦＦが００００に変わっています。

3・10 引き算はこうします

引き算はやさしそうで案外面倒なものです。８０８０系の場合、引き算関係の命令は２つあってＳＵＢとＳＢＢがあります。ＳＵＢはふつうの引き算ですが、ＳＢＢはボローという（借り、桁下がり）をも考慮しての引き算です。まず始めにＳＵＢから勉強していきましょう。

ＳＵＢ
ＳＢＢ

プログラムは簡単でポインタを使っていることだけが複雑そうに見せていますが、＄Ｅ０２０番地の中味から＄Ｅ０２１番地の中味を引いて、＄Ｅ０２２番地にしまっています。ＨＬレジスタをポインタに使っています。５－３を実行する前と実行後のメモリの様子を示しておきます。無事に答は２になっています。Ｘコマンドを使ってレジスタの中味をのぞいてあります。この時ＦというフラグレジスタはＰという正の結果を反映しています。次に３－５を実行する前後のメモリの様子を示しておきました。答は－２ですが、ＦＥとなっています。２の補数表示となっているわけですね。Ｘコマンドを使ってレジスタの中味をのぞいてみますとフラグレジスタはＭという負の結果を反映しています。

16進	2進	
０２	00000010	→＋２
ＦＤ	11111101	→＋２の１の補数
ＦＥ	11111110	→＋２の２の補数

```
E000    21 E020      LXI  H,E020  ←ポインタをセット
E003    7E           MOV  A,M     ←引かれる数をアキュムレータへ
E004    23           INX  H       ←ポイントを１すすめる
E005    96           SUB  M       ←引き算をする
E006    23           INX  H
E007    77           MOV  M,A     ←答をしまう
E008    76           HLT
```

●プログラム実行前後のメモリの様子（５－３の場合）

```
E020 05 03 00 00 00 00 00 00
```

```
E020 05 03 02 00 00 00 00 00
```

●レジスタの様子

```
A :02 F :P----ON- B :0000 D :EDCC H :E022 A':00 F':P----O-- B':8000 D':0080
H':0000 IX:1008 IY:0000 I :F3 PC:E009 SP:DFFB
```

●プログラム実行前後のメモリの様子（３－５の場合）

```
E020 03 05 00 00 00 00 00 00
```

```
E020 03 05 FE 00 00 00 00 00
```

先の計算でＳＵＢだけですんだのは８ビットの計算だったからです。これが16ビットとなりますと、ＳＵＢだけではすまなくなります。８ビットずつ計算すると２回計算しなければならないので、借りが発生することがあるからです。借りはボロー（Ｂｏｒｒｏｗ）といい、キャリーフラグＣの補数をとってあらわすことになっています。これを考慮しないと16ビットの計算がうまくできません。そこで、このための引き算はＳＢＢを使います。プログラムはＨＬレジスタをポインタ１に使い、ＤＥレジスタをポインタ２に使っています。アキュムレータＡに引かれる数の下位をしまい、ポインタ１を２だけすすめて引く数の下位との差を作ってポインタ２の指定する番地にしまっています。この時はＳＵＢでよいのです。理由はわかりますね。

　次にポインタ１を１だけもどして引かれる数の上位をアキュムレータＡにしまい、ポインタ２を２だけすすめて引く数の上位との差を作ってポインタ２の指定する番地にしまって終わりです。

```
  12
 −13
借り 1  FF→答

  FE
 −ED
 − 1
  10

従って FE12
      −ED13
       10FF→答
```

番地	内容	
E000–E012	メインプログラム	
	未使用	
E020	引かれる数	下位
E021		上位
E022	引く数	下位
E023		上位
	未使用	
E030	答	下位
E031		上位
	未使用	

```
E000    21 E020      LXI   H,E020
E003    11 E030      LXI   D,E030
E006    7E           MOV   A,M
E007    23           INX   H
E008    23           INX   H
E009    96           SUB   M
E00A    12           STAX  D
E00B    13           INX   D
E00C    2B           DCX   H
E00D    7E           MOV   A,M
E00E    23           INX   H
E00F    23           INX   H
E010    9E           SBB   M
E011    12           STAX  D
E012    76           HLT
```

●プログラム実行後のメモリの様子

```
E020 12 FE 13 ED 00 00 00 00
E030 FF 10 00 00 00 00 00 00
```

第4章

繰り返しのあるプログラム

4・1　和を計算する

本章からはくり返しのあるプログラム、すなわちループのあるプログラムを勉強します。ＢＡＳＩＣではどうやっていたかを思い出してみましょう。

アドレス	内容
E000	機械語プログラム
E00E	未使用
E020	答
E021	項　数
E022	和をとる項
E02B	
EDFF	

```
100 N=10
110 SUM=0
120 READ A
130 SUM=SUM+A
140 N=N-1
150 IF N>0 THEN GOTO 120
160 PRINT SUM
170 END
180 '
200 DATA 1,2,3,4,5,6,7,8,9,10
```

アセンブリ言語のプログラムでも考え方は同じであるといえます。メモリマップを左に示しました。プログラムを下に示しておきます。

```
E000    21 E021       LXI   H,E021  ←―＄Ｅ０２１をＨＬレジスタに
E003    46            MOV   B,M     ←―レジスタＭの中味をレジスタＢに格納
E004    97            SUB   A       ←―アキュムレータをクリア
E005    23            INX   H       ←―ＨＬレジスタを１進める
E006    86            ADD   M       ←―アキュムレータの中味にレジスタＭを加える
E007    05            DCR   B       ←―レジスタＢの中味、つまり項数を１減らす
E008    C2 E005       JNZ   E005    ←―レジスタＢが０でなければＥ００５へ
E00B    32 E020       STA   E020    ←―アキュムレータの中味を＄Ｅ０２０番地へ
E00E    76            HLT           ←―終わり
```

プログラム実行前のメモリの内容を下図に示します。

```
        ↓―――項数が入っている（この場合10進数で10個）
E020 00 0A 01 02 03 04 05 06 07 08 09 0A
           └―――――――――――――――――――――┘
                       ↑―――和をとる数
```

プログラム実行後のメモリの内容を次頁に示します。＄Ｅ０２２番地から＄Ｅ０２Ｂ番地までの０１，０２，０３，……０Ａまでの和、10進数での55が、16進数の37で＄Ｅ０２０番地に入っています。

```
E020 37 0A 01 02 03 04 05 06 07 08 09 0A
```
↑ 1から10までの和55が16進数で入っている

プログラムの説明をしておきましょう。

まずHLレジスタに＄E021をロードします。＄E021番地には和をとるべき項の数がしまわれています。この場合10個（16進で0A）となっています。この10という値をBレジスタにとりこむために、MOV B, M を使います。この意味はもちろんHLレジスタが指し示している＄E021番地の中味をBレジスタに移動するということです。BASICのプログラムのN＝10にほぼ対応します。

第3行目の SUB A は、アキュムレータをクリア（0に）することです。XRA A でもかまいません。BASICのプログラムでのSUM＝0に対応しています。

第4行目、第5行目の INX H、ADD M でHLレジスタの値を1ずつ増加させ、ポインタの値を1ずつ増加させています。ポインタの指し示すデータを次々にアキュムレータの内容に加えて行きます。BASICのプログラムでの READ A, SUM＝SUM＋A に対応しているといえます。

第6行目の DCR B は項数のカウンタを1ずつ減少させています。N＝N－1に対応します。

第7行目の JNZ E005 は項数のカウンタが0にならない限りは和をとる操作をくり返せということで、BASICのプログラムでの IF N＞0 THEN GOTO 120に相当します。

第8行目の STA E020 はアキュムレータにある和の値を＄E020番地にしまいなさいということです。

アセンブリ言語のプログラムとBASIC言語のプログラムを比較してみると、プログラムの長さは同じ位になっていることに気がつくでしょう。プログラムの実行速度は約1000倍違うのが普通なので、高速性を要求されるときにはアセンブリ言語のプログラムで書いたほうが有利ということになります。

もう一つ注意しておかねばならないのは、ここでの和のプログラムは総和255をこえない場合にだけ適用できるということです。なぜだかわかりますね。

SUB
(SUBTRACT REGISTER OR MEMORY FROM ACCUMULATDR)
● SUB Aはアキュムレータの中味からアキュムレータの中味を引くこと、つまり、ゼロクリアすることです。

XRA
(EXCLUSIVE-OR REGISTER OR MEMORY WITH ACCUMULATOR)
● XRA Aはアキュムレータの中味とアキュムレータの中味の排他的論理和をとる、つまり、これもゼロクリアすることです。

DCR
(DECREMENT REGISTER OR MEMORY CONTENTS)

JNZ
(JUMP IF NOT ZERO)

4·2 16ビットの和の計算法

番地	メモリ	
E000	機械語プログラム	
E016		
	未使用	
E020	和	下位
E021		上位
E022	項　数	
E023	和をとる項	
E02C		
	未使用	

前節の和の計算法では255までの和しかとれませんでした。せめて65535までくらいの和はとれるようにしたいものです。そこでプログラムを書きかえてやることにします。メモリマップを左図に示します。さっそくプログラムを検討することにしましょう。

```
E000    21 E022    LXI   H,E022
E003    46         MOV   B,M
E004    97         SUB   A
E005    4F         MOV   C,A
E006    23         INX   H
E007    86         ADD   M
E008    D2 E00C    JNC   E00C
E00B    0C         INR   C
E00C    05         DCR   B
E00D    C2 E006    JNZ   E006
E010    21 E020    LXI   H,E020
E013    77         MOV   M,A
E014    23         INX   H
E015    71         MOV   M,C
E016    76         HLT
```

まずHLレジスタにＥ０２２を代入し、ポインタの値をＥ０２２にします。次に　MOV B，M　でBレジスタにHLレジスタの指し示している＄Ｅ０２２番地の中味である０Ａ（10進数では10）を移動しています。３行目で　ＳＵＢ　Ａ　で、アキュムレータをクリアしています。４行目の　ＭＯＶ Ｃ，Ａ　でCレジスタもクリアしています。このプログラムではA，B，C，H，Lのレジスタを使うということを理解し、その役割をよくつかんでおいてください。

５，６行目でHLレジスタの値を１すすめ、ポインタの指し示すメモリ番地の内容をアキュムレータに加えます。

JNC
(JUMP IF NO CARRY)

ここで、和をとるたびに桁上げがあったかどうか判定します。何しろアキュムレータは８ビットしかないのですから、255までの数しか扱えないのです。桁（ケタ）上げがあったときには、キャリーと呼ばれる桁上げビットが１になります。加算した結果255を超えたというしるしです。

桁上げがなかった（Ｃ＝０）場合、プログラムの実行は＄Ｅ００Ｃ番地へとびます。桁上げがあった（Ｃ＝１）場合、ＩＮＲ　Ｃ　でCレジスタの値を１だけ増加します。Cレジスタとキャリー Cを混同し

INR
(INCREMENT REGISTER OR MEMORY CONTENTS)

ないで下さい。

どちらのルートをとっても第9行目の　DCR　B　でBレジスタの内容を1へらします。Bレジスタは項数が0でない限り、和の計算がくりかえされます。和をとるべき項がなくなったら、11〜14行目で和の下位をE020番地に、和の上位をE021番地にしまうことになります。

プログラム実行前のメモリの状態を下図に示します。

```
                値数
E020 00 00 0A A1 A2 A3 A4 A5 A6 A7 A8 A9 AA
                  和をとる数
```

プログラム実行後のメモリの状態を下図に示します。$E023番地から$E02C番地までのA1，A2，A3，……，AAの和、0677が$E020，$E021番地に入っています。

```
E020 77 06 0A A1 A2 A3 A4 A5 A6 A7 A8 A9 AA
     A1からAAまでの和677が下位(77)、上位(06)の順で入る
```

ここでSTAと　MOV M, A　の違いについてふれておきましょう。簡単な例をとってみます。

```
STA  E020        32  E0 20
```

ですし、他方、

```
LXI  H, E020     21  20E0
MOV  M, A        77
```

です。

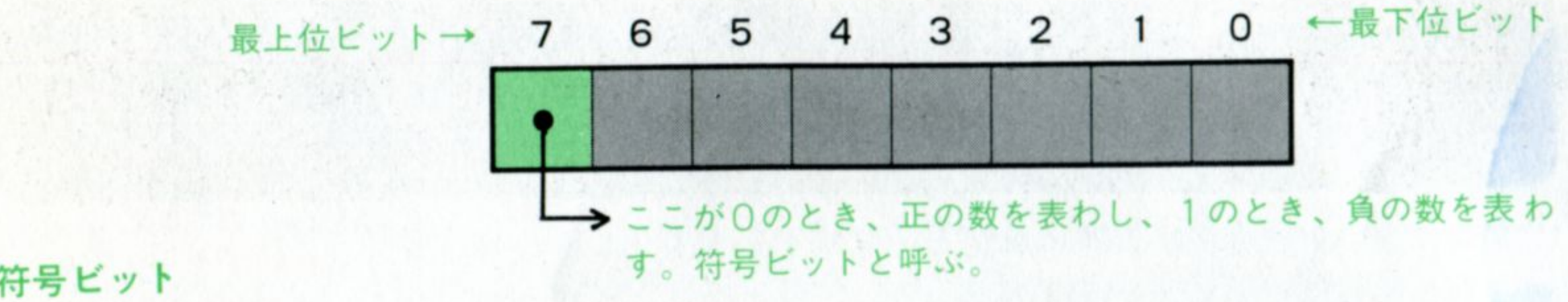
STAは3バイトですむのに対し、MOV M, A　は4バイトかかっています。いつもこうであるならSTAばかり使えばよさそうなものですが、表のような連続した値を扱うときには、MOVは4バイト必要とせず、最初に　LXI H, E020　とポインタを指定さえしておけば、あとは　MOV M, A　の1バイトですみます。従って、

連続したデータを扱うときは、MOV　M, A

孤立したデータを扱うときは、STA

と使いわけるとよいと思います。これは　MOV A, M　とLDAの使いわけに関しても同じことがいえます。

4・3　負の数をかぞえること

２進法で正の数を表現することはやさしいのですが、負の数はどういうふうに表現したらよいでしょうか。よく使われるやり方は、数の最上位の桁のビットを符号として解釈することです。８ビットの場合を考えてみますと次のようになります。

符号ビット

正負は符号ビットが０か１で判断するとしても、負の数自体はどう表現するのでしょうか。結果から見ることにしましょう。

2進(Binary)	10進(Decimal)	16進(Hexadecmal)
10000000	−128	80
10000001	−127	81
10000010	−126	82
10000011	−125	83
⋮	⋮	⋮
⋮	⋮	⋮
11111110	−2	FE
11111111	−1	FF
00000000	0	00
00000001	+1	01
00000010	+2	02
00000011	+3	03
⋮	⋮	⋮
⋮	⋮	⋮
01111101	+125	7D
01111110	+126	7E
01111111	+127	7F

はじめて勉強する人にとってわかりにくいと思われるのは、−1,

－2，……，－127，－128の２進法表現の作り方です。表をじっと見てください。何か規則性があることは確かですが、どうやって作っているのでしょう。それには１の補数、２の補数というものを勉強しなければなりません。

１の補数(ones complement)

１の補数

補数などとわかりにくそうですが、そうでもありません。ある２進数の１の補数というのは、それぞれのビットの数を１から引いて作ることによって実現できます。わかりやすく０と１の場合を考えてみましょう。

０の場合　　１－０＝１……０の１の補数は１

１の場合　　１－１＝０……１の１の補数は０

形式的にいうと０は１におきかえ、１は０におきかえればよいということがわかります。そこで０００００００１の１の補数を作ってみましょう。

２の補数

もとの数　　00000001

１の補数　　11111110

これはきわめて簡単でした。

２の補数 (twos complement)

２の補数は１の補数に１を足すことで作ることができます。

１の補数　　00000001

２の補数　　11111111

実は２の補数を作ることが、２進法で負の数を作ることになっているのです。たとえばいまの例を確かめてみましょう。

＋１の２進表現　　00000001

＋１の２の補数　　11111111　→　表を見ると－１

負の数のとり扱いは実はひき算と多いに関係があります。引き算では２の補数を作って足し算をしています。そこで本節では、負の数はどう作るか、負の数はどう見わけるかだけをよく理解しておいてください。

もう一つ気をつけておいたほうがよいことがあります。それは符号つきの数で表現できる数の範囲です。たとえば８ビットの場合は表からも明らかなように、－128から＋127であることです。－128 から＋128とはなりませんので、よく注意してください。16ビットの場合は、－32768から＋32767の範囲になります。

```
E000    21 E021     LXI  H,E021   ← HLレジスタに$E021を
E003    46          MOV  B,M      ← レジスタMの値をBレジスタへ
E004    3E 7F       MVI  A,7F     ← 7Fをアキュムレータへ
E006    0E 00       MVI  C,00     ← 00をCレジスタへ
E008    23          INX  H        ← HLレジスタの値を1進める
E009    BE          CMP  M        ← アキュムレータの中味から
                                    レジスタMの中味を引く
E00A    D2 E00E     JNC  E00E     ← キャリーフラグが0ならE00Eへ
E00D    0C          INR  C        ← Cレジスタを1ふやす
E00E    05          DCR  B        ← Bレジスタの値(カウンタ)を1へらす
E00F    C2 E008     JNZ  E008     ← カウンタが0でなければE008へ
E012    79          MOV  A,C      ← Cレジスタの中味をアスキュレータへ
E013    32 E020     STA  E020     ← アキュムレータの中味を$E020へ
E016    76          HLT           ← 終わり
```

プログラム実行前のメモリの状態を下図に示します。

```
E000 21 21 E0 46 3E 7F 0E 00 23 BE D2 0E E0 0C 05 C2
E010 08 E0 79 32 20 E0 76 FF 00 FF 00 FF 00 FF 00 FF
E020 00 0A 01 02 03 04 05 F5 F6 F7 F8 F9 FA
```

プログラム実行後のメモリの状態を下図に示します。

メモリマップとてらしあわせてみてください。

```
E000 21 21 E0 46 3E 7F 0E 00 23 BE D2 0E E0 0C 05 C2
E010 08 E0 79 32 20 E0 76 FF 00 FF 00 FF 00 FF 00 FF
E020 05 0A 01 02 03 04 05 F5 F6 F7 F8 F9 FA
```

アドレス	内容
E000〜E016	機械語プログラム
	未使用
E020	負数の個数
E021	正負数の個数
E022〜E02B	正負の数
	未使用

プログラムを見ていきましょう。まずHLレジスタにE021を代入して、ポインタの値を$E021番地にします。次に　MOV B,Mで、BレジスタにHLレジスタの指し示している$E021番地の中味である0Aを移します。(正負数の個数は10個としています。その時々によって正負数の個数は異なります。)

次に、Aレジスタに7Fを代入します。さらに4行目でCレジスタに0を代入しています。次にHLレジスタの値を1すすめ、ポインタの指し示す番地を$E022番地へとすすめます。

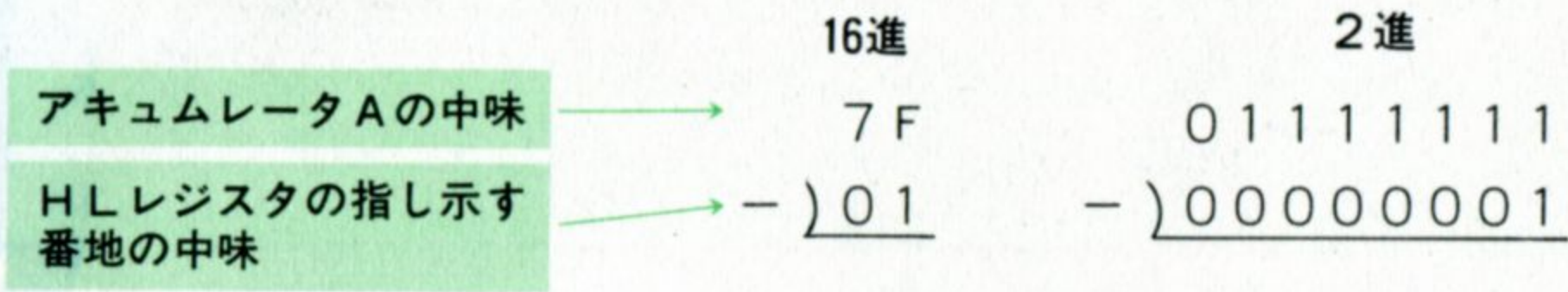
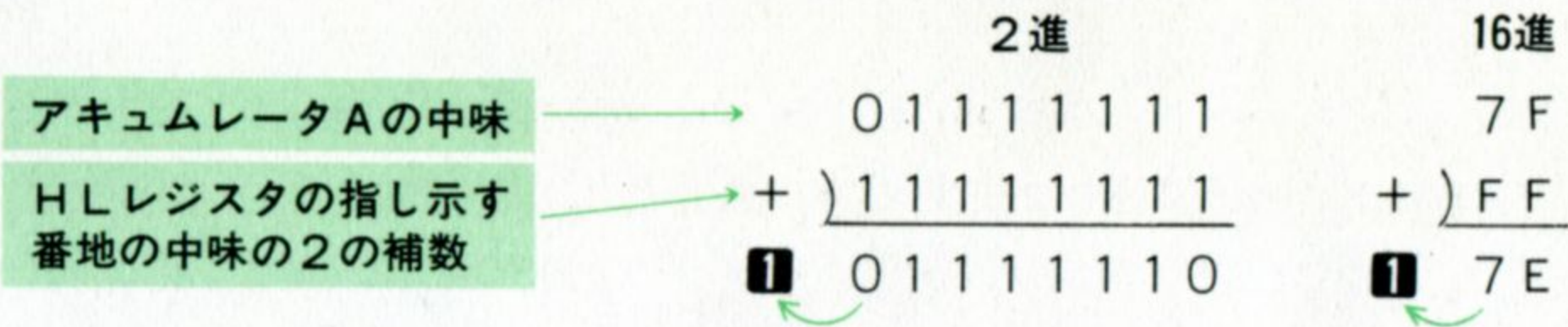
6行目にある　CMP M　は少し説明を要します。CMPというのはCOMPARE REGISTER OR MEMORY WITH ACCMULATORの略語です。だから、CMP M はHLレ

ジスタの指し示す番地の中味とアキュムレータの中味を比較することになります。

CMP M＝（アキュムレータAの中味）
－（HLレジスタの指し示す番地の中味）

CMP

比較した結果を利用するにはフラグというものを利用します。たとえば、HLレジスタが指し示している番地が＄E022のとき、CMP Mを実際にやってみましょう。

	16進	2進
アキュムレータAの中味 →	7F	01111111
HLレジスタの指し示す番地の中味 →	－）01	－）00000001

引き算は先にのべた2の補数を使うのです。だから上の計算は次のようになります。

● 2の補数の作り方は覚えていますか？ 1の補数を作って1を足すのでしたね。忘れた人は前節を見てください。

	2進	16進
アキュムレータAの中味 →	01111111	7F
HLレジスタの指し示す番地の中味の2の補数 →	＋）11111111	＋）FF
	1 01111110	1 7E

7F－01は目でみてすぐにわかるように7Eです。2の補数を使った計算で桁上がり分を無視すれば正しい答がでていることがわかります。桁上がりのことをCarry（キャリー）といいます。

引き算のSUBやSBBやCMPにおいてはキャリーの結果とキャリーフラグの結果は反転して使われることに注意しておく必要があります。キャリーというのはもともと桁上げて、引き算の場合には桁下がり（借り）になります。英語ではボローといいます。インテル8080の場合にはボローのフラグはありませんから、キャリーの補数をとったものが、キャリーフラグCに代入されます。だから上の計算でキャリーが1なら、キャリーフラグCは0となっていることになります。これは大変間違いやすいので注意してください。

結果が正のときの判定法	結果が負のときの判定法
桁上がり(キャリー)フラグ　0	桁上がり(キャリー)フラグ　1
符号（サイン）フラグ　0	符号（サイン）フラグ　1

何ともややこしい話ですが、整理するとこういうことになります。

（アキュムレータＡの中味）≧（ＨＬレジスタの指し示す番地の中味）
→ 桁上がり（キャリー）フラグ　0
　 符号（サイン）フラグ　0

（アキュムレータＡの中味）＜（ＨＬレジスタの指し示す番地の中味）
→ 桁上がり（キャリー）フラグ　1
　 符号（サイン）フラグ　1

ここら辺がアセンブリ言語のプログラムの最初の難関となりますので、わからない場合は一服し、よくふり返ってみてください。はじめての人にはそうやさしくはないと思います。慣れた人は簡単というでしょうが、それほどやさしくはないので、わからなくとも心配する必要はありません。

つまりキャリーフラグＣは、ボロー（借り）が必要になる場合には１、ボロー（借り）が必要でない場合には０となります。

ここでのプログラムは面白いテクニックを使っています。それは、７Ｆから負の数を引いた場合と、７Ｆから正の数を引いた場合を考えるとわかることです。

❶ 7F−(−1)の計算

```
   01111111……7F
− )11111111……(−1)
       ↓
   01111111……7F
+ )00000001……(−1)の2の補数
[0] 100000000……80
```

❷ 7F−1の計算

```
   01111111……7F
− )00000001……1
       ↓
   01111111……7F
+ )11111111……1の2の補数
[1] 01111110……7E
```

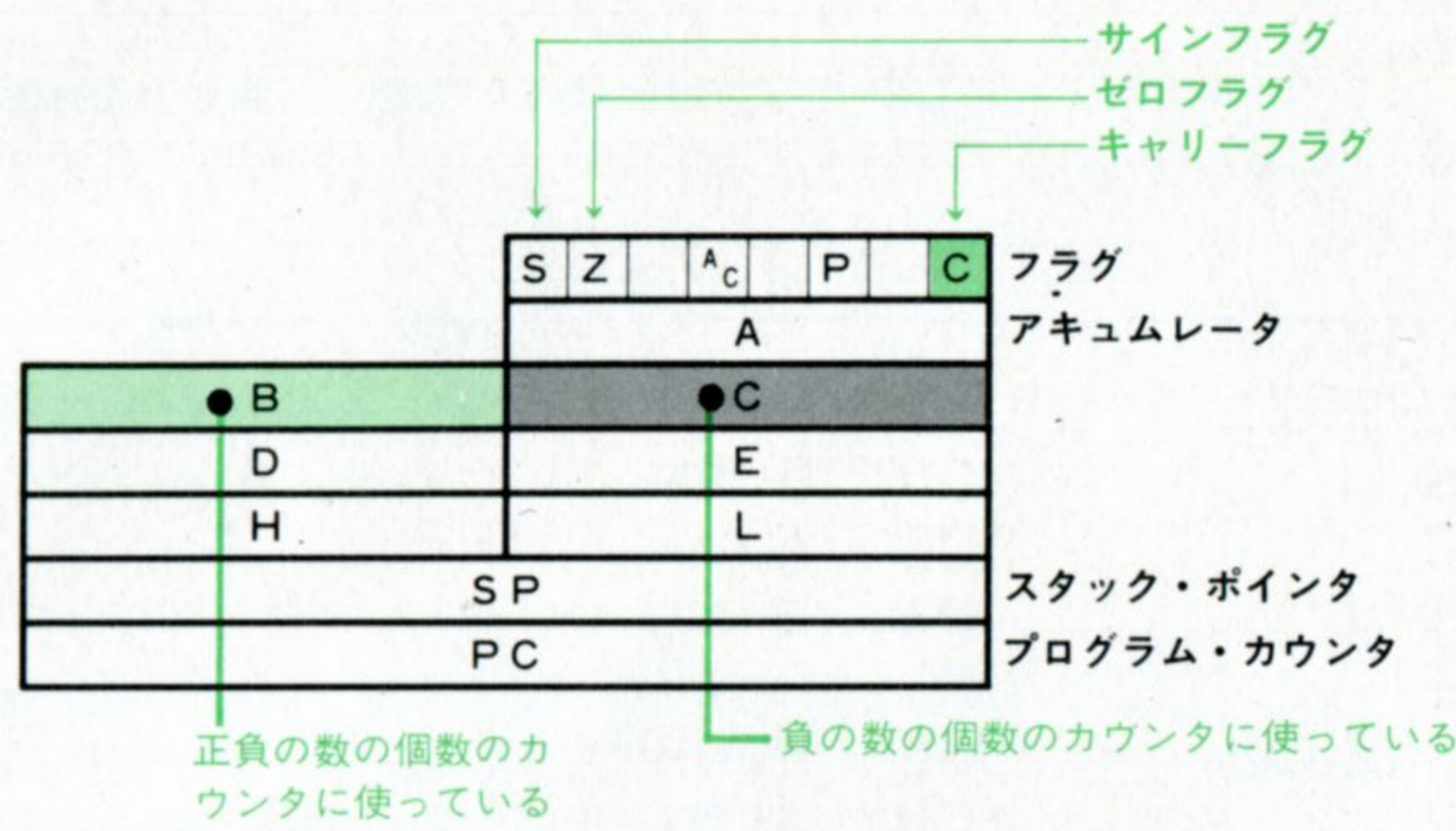
つまり７Ｆと負の数を比較すると桁上げを生じないので、キャリーフラグは１となり、７Ｆと正の数を比較すると桁上げを生ずるのでキャリーフラグは０となります。ＣＭＰ命令を使いますと負の数のときにはキャリーフラグＣは１にセットされ、正の数のときにはキャリーフラグＣは０にリセットされたままということになります。

これだけの準備がありますとプログラムは読みやすくなるでしょう。つまり6行目の　CMP M　で負の数ならばキャリーフラグが1にセットされます。7行目にあるJNCは、JUMP　IF　NO　CARRYですから、負の数の場合はキャリーがありますので8行目にすすみます。

サインフラグ
ゼロフラグ
キャリーフラグ

S Z A_C P C	フラグ
A	アキュムレータ
B / C	
D / E	
H / L	
SP	スタック・ポインタ
PC	プログラム・カウンタ

正負の数の個数のカウンタに使っている

負の数の個数のカウンタに使っている

8行目の　INR Cは、INCREMENT　REGISTER　Cですので、レジスタCの中味を1だけ増加することになります。レジスタCは負の数の個数をかぞえるカウンタになっているのです。

9行目の　DCR Bは、正負の数の個数を数えるカウンタとして使われているレジスタBの値を1だけ減少させます。1つ処理が終了したのですから、当然カウンタの値を1だけ減少させておかねばならないのです。

6行目において、正の数であった場合にはキャリーフラグは0にリセットされたままですので、7行目にある　JNC E00E　にひっかかり、レジスタCの値を増加させることなくE00Eにすすみます。9行目の　DCR Bは正負の数の処理を1つ終えたわけですから、当然1だけ減少させます。

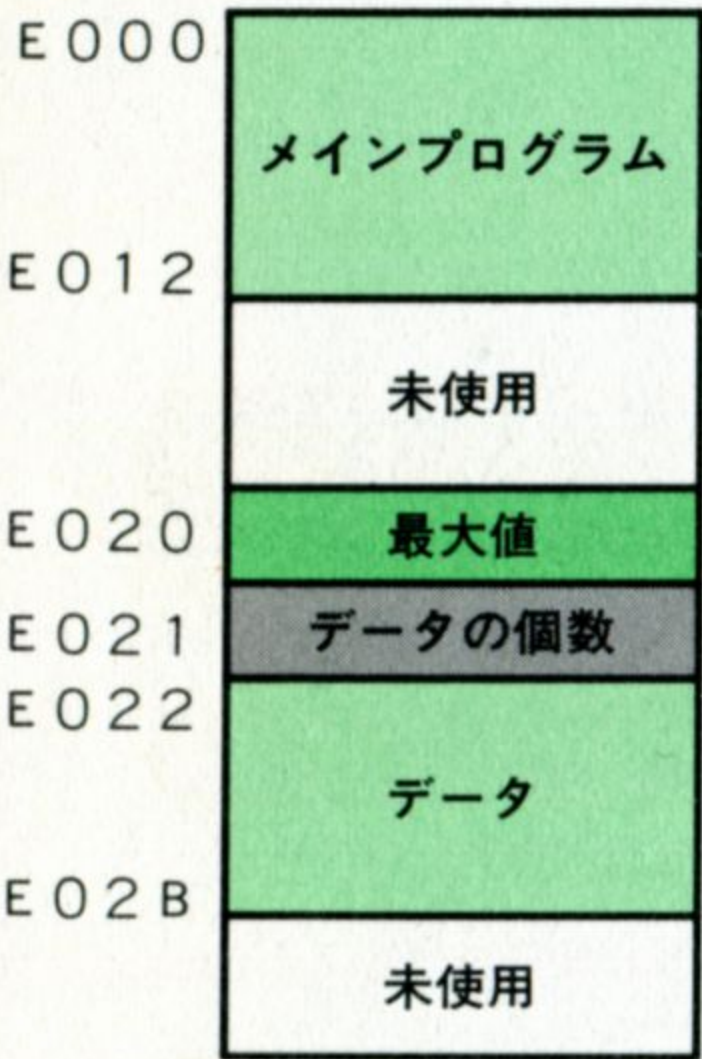
10行目の　JNZ E008はすべての正負の数の処理が終了したかどうかを判定するものです。Bレジスタの正負の数の個数が0になっていないかぎり、$E008番地へ飛んで処理をくり返させます。Bレジスタの値が0になって処理が終了したことになりますと、11行目の　MOV A, C　でCレジスタの負数の個数をアキュムレータAに移し、12行目の　STA E020で、結果を$E020番地にしまいます。これで終わりです。

4・4　いちばん大きい数をみつける

こんどは数の集団の中から一番大きいものをみつけることにしましょう。だいぶなれてきたと思いますので説明は簡単にすることにします。まず、数のデータは＄Ｅ０２２番地からしまっておくことにして、データの終わりには特別な記号を使ったりせず、＄Ｅ０２１番地にデータの個数を入れておき、データの大きさを明示することにします。

プログラムを見ていきましょう。

```
E000   21 E021   LXI  H,E021  ← Ｅ０２１をＨＬレジスタへ
E003   46        MOV  B,M     ← レジスタＭの値をレジスタＢへ
E004   97        SUB  A       ← アキュムレータをゼロクリアする
E005   23        INX  H       ← ＨＬレジスタの値を１すすめる
E006   BE        CMP  M       ← アキュムレータとレジスタＭの値を比較
E007   D2 E00B   JNC  E00B    ← キャリーが０のときジャンプ
E00A   7E        MOV  A,M     ← レジスタＭの値をアキュムレータへ
E00B   05        DCR  B       ← データ数を１へらす
E00C   C2 E005   JNZ  E005    ← データ数が０でなければＥ００５へ
E00F   32 E020   STA  E020    ← アキュムレータの中味を＄Ｅ０２０番地へ
E012   76        HLT          ← 終わり
```

番地	内容
Ｅ０００〜Ｅ０１２	メインプログラム
	未使用
Ｅ０２０	最大値
Ｅ０２１	データの個数
Ｅ０２２〜Ｅ０２Ｂ	データ
	未使用

まずＨＬレジスタにＥ０２１を持ってきます。これによってポインタが＄Ｅ０２１番地を示すことになりました。次に2行目の　ＭＯＶ　Ｂ, Ｍでレジスタ Ｂにポインタの指し示している番地（＄Ｅ０２１番地）の中味である０Ａを持ってきます。Ｂレジスタはデータの個数を示すことになります。第3行目の　ＳＵＢ　Ａ　はアキュムレータＡをクリア（０にすること）するためのもので、ＸＲＡ　Ａ　などとしてもかまいません。次に第4行目の　ＩＮＸ　Ｈ　でポインタの値を1だけすすめてやります。第5行目の　ＣＭＰ　Ｍ　はアキュムレータＡの中味とポインタの指している番地の中味を比較しており、

（アキュムレータＡの中味）－（ポインタの指している番地の中味）

という計算を実行します。

何回か説明してきましたように、キャリーフラグＣは次のようになります。

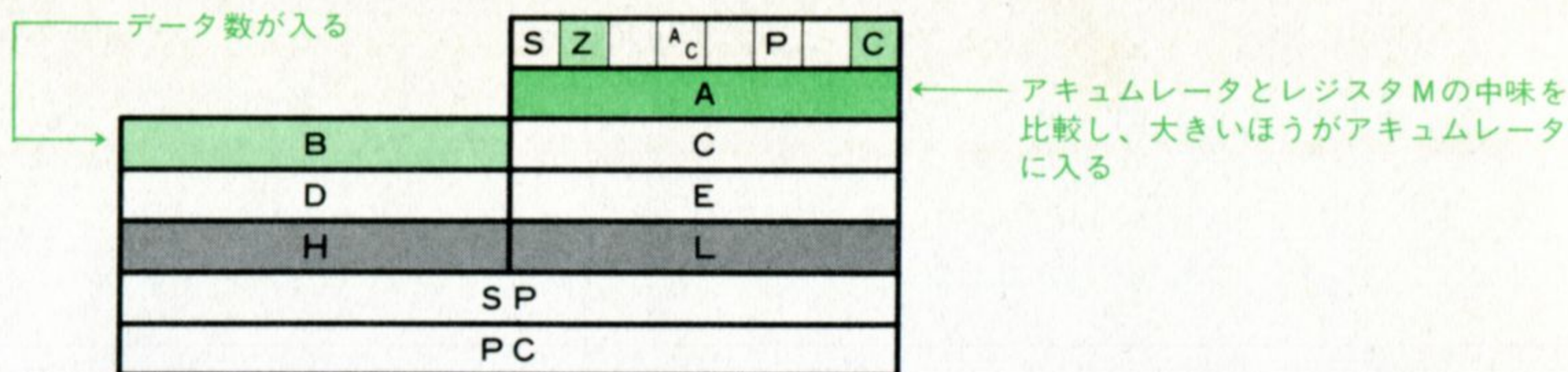
❶　（アキュユレータＡの中味）≧（ポインタの指している番地の中味）のとき　→　キャリーフラグＣは０

❷　（アキュムレータＡの中味）＜（ポインタの指している番地の中味）のとき　→　キャリーフラグＣは１

データ数が入る

S	Z		AC		P		C

	A
B	C
D	E
H	L
SP	
PC	

アキュムレータとレジスタMの中味を比較し、大きいほうがアキュムレータに入る

ですから、アキュムレータAの中味がポインタの指している番地の中味より大きいか等しいとき（これを〝小さくないとき〟と表現することがあります）にはキャリーフラグは0で、6行目の　JNC E00B によって、データの数を1へらして9行目の　JNZ E005 でループを回りつづけます。いまの場合は言葉を変えていうと、アキュムレータAに入っている最大値がデータより大きいか等しい、すなわち小さくない場合です。

アキュムレータAに入っている最大値がデータより小さい場合にはキャリーフラグが1となって7行目の　MOV A, M によってアキュムレータAの最大値が現在のデータと入れ代わります。このときにも8行目の　DCR B でデータの数を1へらして9行目の　JNZ E005 でループを回りつづけます。

データの数が0になりますと9行目の　JNZ E005 には引っかからなくなり、処理が終了します。10行目のSTA E020 によってアキュムレータAに入っている最大値が$E020番地にうつされます。

プログラム実行前のメモリの状況は次の通りです。

```
E000 21 21 E0 46 97 23 BE D2 0B E0 7E 05 C2 05 E0 32
E010 20 E0 76 32 20 E0 76 FF 00 FF 00 FF 00 FF 00 FF
E020 00 0A 01 02 03 04 05 06 07 08 09 0A
```

（0A：データ数、01〜0A：データ）

プログラム実行後のメモリの状況は次のようになります。

```
E000 21 21 E0 46 97 23 BE D2 0B E0 7E 05 C2 05 E0 32
E010 20 E0 76 32 20 E0 76 FF 00 FF 00 FF 00 FF 00 FF
E020 0A 0A 01 02 03 04 05 06 07 08 09 0A
```

（E020：01〜0Aの最大値0Aが入っている）

さて、ここでアセンブリ言語のプログラムは必ずしも面倒なものではないことを示すために、ＢＡＳＩＣで書いたプログラムを示しておきましょう。アセンブリ言語のプログラムと対応がつきやすいようにしたので少し不細工な所がありますが、結構手数がかかることがおわかりのことと思います。

```
100 DIM D(10)
110 FOR I=0 TO 10     ┐
120 READ D(I)         ├── データを配列に読み込む
130 NEXT I            ┘
140 '
150 I=0
160 N=D(I) ←── データ数をカウンタ用の変数Nに代入
170 MAX=0 ←── 最大値を初期化する
180 I=I+1
190 IF MAX-D(I)>=0 THEN GOTO 210 ←── 新しい最大値があらわれなければ２１０行へ
200 MAX=D(I) ←── 最大値を新しく変数ＭＡＸに代入
210 N=N-1
220 IF N<>0 THEN GOTO 180 ←── データ数が０でなければ180行からくり返す
230 PRINT " MAX=";MAX
240 END
250 '
300 DATA 10,1,2,3,4,5,6,7,8,9,10 ←── データ
```

4・5　数の正規化について

数の正規化とはききなれない言葉ですが、科学技術計算では、

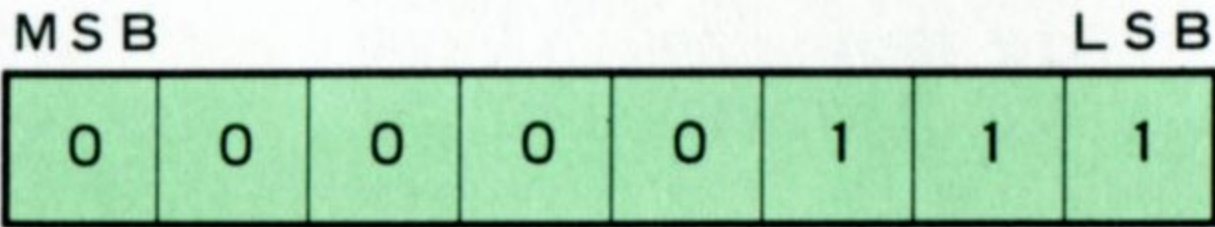
　　０．０００１２３４

とは表現せずに、

　　＋１．２３４×10^{-4}

などと表現します。この場合 ＋１．２３４を仮数部とよび、－４を指数部と呼んでいます。０．０００１２３４を ＋１．２３×10^{-4}という形式へ変換するためには ０．０００１２３４を １．２３４という形式に変換し、いくつ左へ動かしたかを数えねばなりません。これと同じような操作をするのが次のプログラムです。

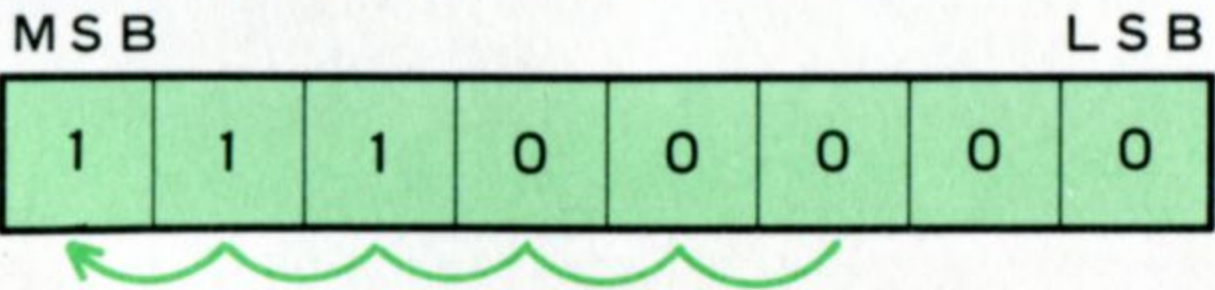
```
E000    97          SUB   A       ← アキュムレータをクリア
E001    47          MOV   B,A     ← レジスタBをクリア
E002    21 E020     LXI   H,E020
E005    86          ADD   M       ← $E020番地の中味とアキュムレータの中味を加えてアキュムレータへ
E006    CA E011     JZ    E011    ← 演算結果が0ならE011へ
E009    FA E011     JM    E011    ← 最上位ビットが1ならE011へ
E00C    04          INR   B       ← シフト回数のカウンタをプラス1
E00D    87          ADD   A       ← アキュムレータの中味を左へ1ビットシフトする
E00E    C3 E009     JMP   E009
E011    23          INX   H
E012    77          MOV   M,A     ← 正規化された数をしまう
E013    23          INX   H
E014    70          MOV   M,B     ← シフト回数をしまう
E015    76          HLT
```

たとえば０７という16進表記の数字を２進法８ビットで表現しますと次のようになります。

MSB
(MOST SIGNIFICANT BIT――最上位ビット)

LSB
(LEAST SIGNIFICANT BIT――最下位ビット)

MSB							LSB
0	0	0	0	0	1	1	1

これを正規化するには、次のようにします。

MSB							LSB
1	1	1	0	0	0	0	0

第１行目の　ＳＵＢ　Ａ　はアキュムレータAを０にします。第２行目の　ＭＯＶ　Ｂ，Ａ　はBレジスタを０にリセットしています。第３行目の　ＬＸＩ　Ｈ，Ｅ０２０　は、ポインタを＄Ｅ０２０番地にしています。第４行目の　ＡＤＤ　Ｍ　はアキュムレータAにポインタの指

し示している番地の中味（つまり正規化されるべきデータ）を加えています。

JZ
(JUMP IF ZERO——ゼロフラグがたっていればジャンプする)

JM
(JUMP IF SIGN MINUS——サインフラグがたっていればジャンプ)

第5行目の　JZ E011 はデータが0なら、処理する必要がありませんから＄E011番地にとばしています。

第6行目の　JM E011 で最上位ビット（MSB）に1が入ったかどうかを見ています。（このプログラムでは符号の＋，－は扱いません。1．234×10^{-4}というような形式への処理だけを考えています。＋1．234×10^{-4}というような形式への処理を考えるにはもう少し工夫が必要です。）1が入れば処理終了ですし、入っていなければ処理を続行します。

第7行目の　INR B はBレジスタの値を1ふやし、左へ1シフトすることを意味しています。第8行目の　ADD A は前に勉強したように、

（アキュムレータAの中味）＋（アキュムレータAの中味）
→（アキュムレータAの中味）

とすることですから、アキュムレータAの中味を2倍することになり、結局左へ1ビットシフトすることになります。もしわからない場合には第3章の2節をもう一度読み直してください。

第9行目の　JMP E009 は強制的に＄E009番地へとばすものです。ハードウエア的にいいますと、プログラムカウンタ（PC）の値をE009に変更し、プログラムの実行の流れを変更しています。E009とE00Aの間はループを作っていることがわかります。

第10行目の　INX H はポインタの値を1ふやし＄E021番地へとすすめています。

第11行目の　MOV M, A でアキュムレータAの中味（このときすでにデータは正規化されている）をポインタの指し示す＄E021番地へすすめています。

第12行目の　INX H で再びポインタを＄E022番地へとすすめ、第13行目の　MOV M, B でBレジスタにしまわれているシフト回数を＄E022番地にしまいます。これでプログラムは終わりです。

何とも面倒でかなわないとおっしゃる方もあるかも知れませんが、BASICで作ったプログラムと比較してください。

```
100 A=0
110 B=A
120 READ A
130 IF A=0 THEN GOTO 200
140 IF (A AND &H80)=&H80 THEN GOTO 200
150 B=B+1
160 A=A+A
170 GOTO 140
200 PRINT "NORMALIZED DATA = ";HEX$(A)
210 PRINT "SHIFT NUMBER =";B
220 END
240 DATA 7
```

結構わずわらしいものでしょう。

さてデータとして01を$E020番地にしまっておいた場合のメモリの変化の状況を図に示しておきます。

プログラム実行前のメモリのようす。

```
E020 01 00 00
```

プログラム実行後のメモリのようす

```
E020 01 80 07
         ↑  ↑
         |  └──── シフトされた回数
         └─────── 正規化された値
```

皆さんもデータを入れかえていろいろためしてみてください。

4・6　ASCII符号とは何だろう

ASCII符号

これからパソコンを勉強していく上で、どうしても覚えておいていただきたいものにASCII（アスキー）符号があります。パソコンと周辺機器との間のデータのやりとりや、パソコン内部でのデータ処理はASCII符号が使われることが多いので大切になるのです。

たとえば日電純正のプリンタをお持ちの方は次のようにしてみてください。

LPRINT　CHR$（10）

すると1行送り出されるでしょう。表を見ますと10進法の10は16進法では0Aでするから、横の0と縦のAのまじわる所の記号をみますと、LFがみつかります。LFはLine Feed(改行)の略号です。

さらに次のようにしてみてください。

LPRINT　CHR$（13）

今度も1行送り出されたでしょう。10進法の13は16進法では0Dですから、横の0と縦のDのまじわる所の記号をみつけますとCRがみつかります。CRはCarriage Return(復帰) の略号です。

もう一つやってみましょう。

LPRINT　CHR$（12）

用紙が一枚送り出されたでしょう。10進法の12は16進法の0Cですから、FFという記号が対応することがわかります。FFはForm Feedの略号です。

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE		0	@	P	`	p	▁	┴		ー	タ	ミ	═	×
1	SOH	DC1	!	1	A	Q	a	q	▂	┬	。	ア	チ	ム	╞	円
2	STX	DC2	"	2	B	R	b	r	▃	┤	「	イ	ツ	メ	╪	年
3	ETX	DC3	#	3	C	S	c	s	▄	├	」	ウ	テ	モ	╡	月
4	EOT	DC4	$	4	D	T	d	t	▅	▔	、	エ	ト	ヤ	◢	日
5	ENQ	NAK	%	5	E	U	e	u	▆	─	・	オ	ナ	ユ	◣	時
6	ACK	SYN	&	6	F	V	f	v	▇	│	ヲ	カ	ニ	ヨ	◥	分
7	BEL	ETB	'	7	G	W	g	w	█	▕	ァ	キ	ヌ	ラ	◤	秒
8	BS	CAN	(	8	H	X	h	x	▏	┌	ィ	ク	ネ	リ	♠	
9	HT	EM	)	9	I	Y	i	y	▎	┐	ゥ	ケ	ノ	ル	♥	
A	LF	SUB	*	:	J	Z	j	z	▍	└	ェ	コ	ハ	レ	♦	
B	VT	ESC	+	;	K	[	k	{	▌	┘	ォ	サ	ヒ	ロ	♣	
C	FF	FS	,	<	L	¥	l	¦	▋	╭	ャ	シ	フ	ワ	●	
D	CR	GS	-	=	M	]	m	}	▊	╮	ュ	ス	ヘ	ン	○	
E	SO	RS	.	>	N	^	n	~	▉	╰	ョ	セ	ホ	゛	╱	
F	SI	US	/	?	O	_	o		┼	╯	ッ	ソ	マ	゜	╲	

この表をみるといろいろ面白いことがわかるのです。たとえば数字の0から9は16進法の30から39に対応し、英語の大文字のアルファ

ベットＡ～Ｚは41から５Ａに対応し、小文字は61から７Ａに対応しています。

さらに80からＦＦは本来のＡＳＣＩＩ符号にはないのですが、カナが使えるように拡張した体系になっています。カナのアからワンまではＢ１からＤＤに対応していることがわかります。80からＦＦまでのわりあて法はパソコンやプリンタによって少しずつ違いがあるようです。

このＡＳＣＩＩ符号が２進、10進、16進のいずれでも自由自在に頭に浮かんで来るようになっていただきたいのですが、最初のうちは表をみながらで十分です。

前ページのＡＳＣＩＩ符号の表を打ち出すためのＢＡＳＩＣプログラムを下図に示しておきました。

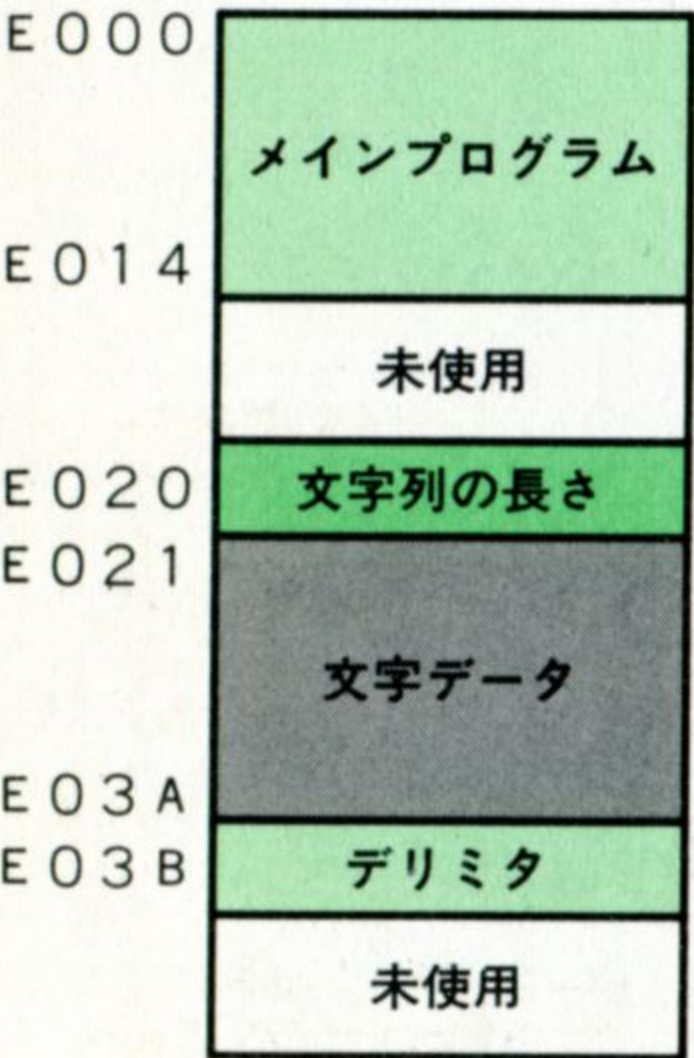
```
100     A$="NULSOHSTXETXEOTENQACKBEL"
110 A$=A$+"BS HT LF VT FF CR SO SI "
120 A$=A$+"DLEDC1DC2DC3DC4NAKSYNETB"
130 A$=A$+"CANEM SUBESCFS GS RS US "
140 WIDTH 80,25
150 SCREEN 0,0
160 LOCATE 3,0
170 FOR N=0 TO 15
180 PRINT "    ";HEX$(N);
190 NEXT N
200 PRINT:PRINT
210 FOR M=0 TO 15
220 PRINT HEX$(M);"   ";
230 FOR N=0 TO 15
240 C=N*16+M
250 IF C>31 THEN GOTO 290
260 C$=MID$(A$,3*C+1,3)
270 PRINT " ";C$;
280 GOTO 310
290 C$=CHR$(C)
300 PRINT "   ";C$;
310 NEXT N
320 PRINT
330 NEXT M
340 END
```

4・7 文字列の長さをかぞえる

文字列というのは文字の並びのことです。あまりパッとしないテーマだとお考えの方もあるかも知れませんが、そんなことはないのです。コンピュータを利用する上においてはもっとも大切なテーマであるといえます。まず文字列の長さを計算することから始めましょう。

メモリのある番地から始まって、ある番地まで文字がつまっているとします。文字列の終わりはデリミタというもので表示することにします。この例題ではデリミタを０Ｄすなわち１３で示すことにしています。前節で説明したＡＳＣＩＩコードでは　ＣＨＲ＄（１３）＝ CR となっています。Ｃａｒｒｉａｇｅ　Ｒｅｔｕｒｎ　とは復帰のことです。

```
E000    21 E021     LXI   H,E021
E003    06 00       MVI   B,00      ← Bレジスタをクリア
E005    3E 0D       MVI   A,0D      ← アキュムレータにデリミタを入れる
E007    BE          CMP   M         ← レジスタMの中味がデリミタと一致するかどうか比較
E008    CA E010     JZ    E010      ← 演算結果がゼロならデリミタと一致したことだからループから抜ける
E00B    04          INR   B
E00C    23          INX   H
E00D    C3 E007     JMP   E007
E010    78          MOV   A,B       ← 文字列の長さをアキュムレータへ
E011    32 E020     STA   E020      ← アキュムレータの中味を＄Ｅ０２０番地へ
E014    76          HLT
```

番地	内容
E000	メインプログラム
E014	未使用
E020	文字列の長さ
E021	文字データ
E03A	デリミタ
E03B	未使用

プログラムをみていきましょう。まず第１行目でポインタの値をＥ０２１にしています。第２行目でＢレジスタに０を入れます。Ｂレジスタは文字列の長さを数えるために使われています。第３行目でアキュムレータＡに０Ｄを入れています。０Ｄはデリミタをあらわします。第４行目でポインタの指し示す番地の中味とアキュムレータＡの中味を比較しています。比較しているものがデリミタ０Ｄであれば第５行目の　ＪＺ　Ｅ０１０　で＄Ｅ０１０番地にとびますが、比較しているものがデリミタでない場合には第６行目でＢレジスタの値を１だけふやし、第７行目でポインタの値を１だけふやします。第８行目の　ＪＭＰ　Ｅ００７　は無条件に＄Ｅ００７番地にジャンプしなさいということです。第９行目の　ＭＯＶ　Ａ，Ｂ　には、５行目の　ＪＺ　Ｅ０１０　からしかとんでこられません。つまり文字列の終わりをあらわすデリミタにあった場合だけしかとんでこれないのです。　ＭＯＶ　Ａ，Ｂ　でＢレジスタに入っている文字列の長さがアキュムレータＡにうつ

されます。第10行目のＳＴＡ　Ｅ０２０ で文字列の長さが＄Ｅ０２０番地にしまわれます。これで終わりです。

```
100 DIM D$(40)
110 FOR I=1 TO 40
120 READ D$(I)
130 NEXT I
140 '
150 B=0
160 A$="*"
170 I=1
180 IF D$(I)=A$ THEN GOTO 220
190 B=B+1
200 I=I+1
210 GOTO 180
220 PRINT " THE LENGTH OF THE STRING IS = ";B
230 END
240 DATA A,B,C,D,E,F,G,H,I,J
250 DATA K,L,M,N,O,P,Q,R,S,T
260 DATA U,V,W,X,Y,Z,*,0,0,0
270 DATA 0,0,0,0,0,0,0,0,0,0
```

ＢＡＳＩＣでも同じようなことができますのでやってみました。デリミタは＊を使っています。ＣＨＲ＄（１３）をデリミタにしてもよいのですが、プログラムが読みにくくなるので＊で代用しました。

下図にプログラム実行前後のメモリの様子を示しておきます。右側をみるとＣＨＲ＄（４１）からＣＨＲ＄（５Ａ）が英語のアルファベットの大文字のＡ，Ｂ，Ｃ，……，Ｚに対応していることがわかるでしょう。またプログラム実行後の＄Ｅ０２０番地には１Ａという値が入っています。16進数の１Ａを10進数に直すと26です。アルファベット26文字を数えさせたところ、正しく26と数えてきました。

プログラム実行前のメモリの様子

ここからＳＥ０３Ｂ番地までアルファベットがＡＳＣＩＩ符号で入っている

```
E020 00 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
E030 50 51 52 53 54 55 56 57 58 59 5A 0D
```

プログラム実行後のメモリの様子

10進数の26

```
E020 1A 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
E030 50 51 52 53 54 55 56 57 58 59 5A 0D
```

ついでに1つ応用問題を考えてみました。マイコンの世界では、文字列のデリミタとして使われるものに次の3種類があって混乱しています。

❶ (CR) だけのもの (16進では0D)

❷ (CR), (LF) と続くもの (16進では0D, 0A)

❸ (LF) だけのもの (16進では0A)

そのためプログラムでは、これらのどれがデリミタとして使われている場合でも文字列の長さを数えられるようにしておくことが望ましいのです。そこで、これをプログラムしてみましょう。実は❶と❷は同じタイプで (CR) だけを読んだら終わりにさせてしまいます。❷の場合 (LF) が余っているのが問題になる場合がありますが、文字列の長さを数えるだけなら問題はありません。❸の場合には (LF) を読んだら終わりになるようにします。❶〜❸の全ての場合に対応できるようにしたのが下のプログラムです。

```
E000    21 E021     LXI  H,E021
E003    06 00       MVI  B,00
E005    3E 0D       MVI  A,0D
E007    BE          CMP  M
E008    CA E016     JZ   E016  ←レジスタMとデリミタ0Dが等しければE016へ
E00B    3E 0A       MVI  A,0A
E00D    BE          CMP  M
E00E    CA E016     JZ   E016  ←レジスタMとデリミタ0Aが等しければE016へ
E011    04          INR  B
E012    23          INX  H
E013    C3 E005     JMP  E005  ←デリミタ0D, 0Aにあわない場合強制的にE005へジャンプ
E016    78          MOV  A,B
E017    32 E020     STA  E020
E01A    76          HLT
```

デリミタに (LF) (=CHR$(10)) を使った場合。

```
E020 1A 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
E030 05 51 52 53 54 55 56 57 58 59 5A 0D
```

デリミタに (CR) (=CHR$(13)) を使った場合。

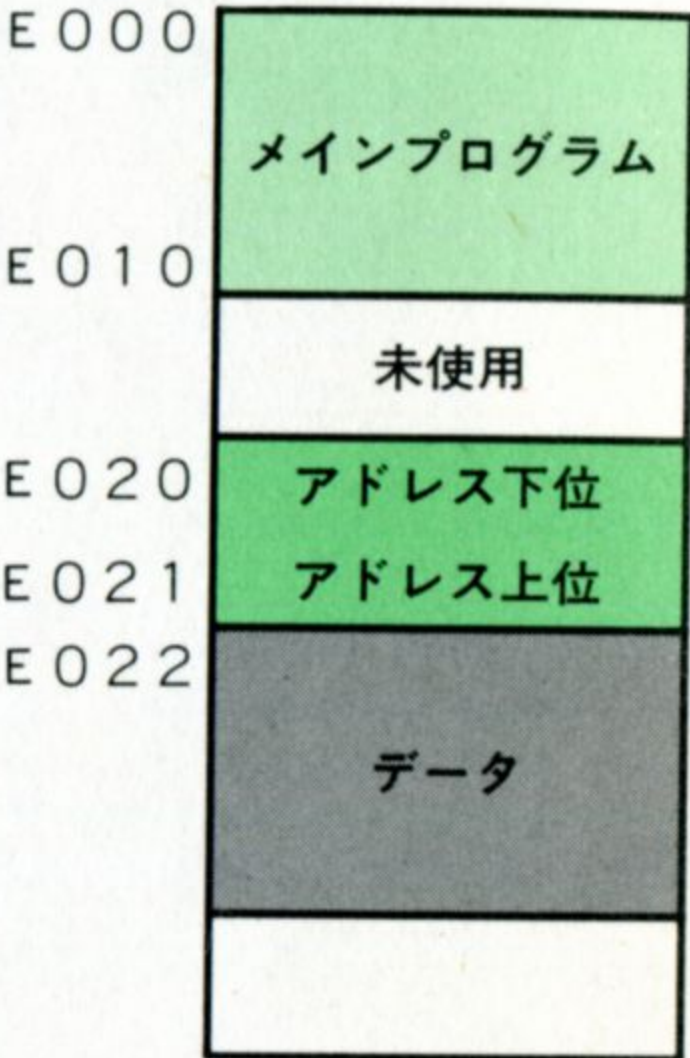
```
E020 1A 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
E030 05 51 52 53 54 55 56 57 58 59 5A 0A
```

4・8　最初の空白でない文字を探す

空白というのはASCIIコードで20にあたります。空白だから何もないのだということにはなりません。この辺が何とも面白いところです。メモリの中に空白のコードがずっと入っていて、ある番地から意味のあるデータが入っていることがよくあります。どこから意味のあるデータが入っているかを探すのが次に示すプログラムです。

まず1行目の　LXI H, E022 でポインタの値を$E022番地にします。2行目の　MVI A, 20 でアキュムレータAに20という空白のコードを入れます。3行目の　CMP M で$E022番地のデータと空白のコードとを比較します。4行目の　JNZ E00D で空白コードと一致しなければE00Dに飛び、空白コードと一致するならば5行目にすすみます。5行目の　INX H はポインタの値を1すすめます。6行目の　JMP E005 は無条件に$E005番地へ飛べということです。

7行目の　SHLD E020 には4行目の　JNZ E00D からしか飛んで来ることができません。SHLDはSTORE H, L REGISTERS DIRECTであり、HLレジスタの中味を$E020, $E021番地にしまいます。$E020番地にはアドレスの下位部分が入り、$E021番地にはアドレス番地の上位部分が入ります。8行目のHLTで終わりです。

番地	内容
E000	メインプログラム
E010	未使用
E020	アドレス下位
E021	アドレス上位
E022	データ

```
E000    21 E022     LXI  H,E022
E003    3E 20       MVI  A,20   ←── $20をアキュムレータへ
E005    BE          CMP  M      ←── レジスタMの値と20を比較
E006    C2 E00D     JNZ  E00D   ←── 演算結果が0でなければジャンプ
E009    23          INX  H
E00A    C3 E005     JMP  E005
E00D    22 E020     SHLD E020   ←── 目的の文字のあった番地をE020、
                                    E021に下位、上位の順で格納
E010    76          HLT
```

プログラムの実行前後のメモリの様子を下図に示してあります。$E027番地に41という20と違うコードが入っているので、$E020、$E021番地にはにはプログラム実行後に27E0が入っています。

プログラム実行前のメモリの様子

```
E020 00 00 20 20 20 20 20 41
```

プログラム実行後のメモリの様子

```
E020 27 E0 20 20 20 20 20 41
```

ここで述べたようなテクニックは現実の局面においては、きわめて大切です。たとえば紙テープの読み始めの部分や磁気テープの読み出しの先頭部分をさがし出すためなどに用いられます。よくあるのは空白の列に続いてＣＲとＬＦのコードが何回か繰り返して入っていたり、ＳＴＸコードが入っていたりします。計測データの処理や暗号解読にとっても重要です。

第5章

文字列のテクニック

5・1　パリティをつけること

アドレス	内容
E000	メインプログラム
E011	
	未使用
E020	データの個数
E021	データ
E02A	
	未使用

最近のパソコンでは、ほとんど考えられなくなりましたが、少し前までは、パソコンにデータを読み込ませようとすると、かなりエラーがでて信頼性に乏しかったものです。カセットとパソコンの間ならともかく、遠距離間のパソコン同士をつなごうとしますと、現在でもいろいろエラーのでることが予想されます。こうしたエラーは大敵です。なぜなら間違ったデータが入力されると間違った結果を生むからです。そこで入力したデータに間違いがある場合には間違いがわかると便利です。このような間違いの検出法の一つにパリティ・チェックというものがあります。これはデータを表現する２進符号の中で使われている０なり１の数を必ず奇数個か偶数個にしておき、もし万一奇数個か偶数個でない時にはエラー発生を知ることができるというものです。たとえばＡＳＣＩＩコードで英文字を表現するには７ビットあれば十分でしたから、８ビットのパソコンでＡＳＣＩＩコードの英文字を扱う場合には最上位ビット（ＭＳＢ）の第８ビットがあまっています。そこで第８ビットをパリティ・チェック用に利用することができます。本節でのプログラムではパリティをつけることにしましょう。

```
E000    21 E020     LXI   H,E020
E003    46          MOV   B,M    ←— データ数をBレジスタへ
E004    0E 80       MVI   C,80   ←— ８０をCレジスタへ
E006    23          INX   H      ←— HLレジスタを１すすめる
E007    7E          MOV   A,M    ←— データをアキュムレータへ
E008    B1          ORA   C      ←— ２数の論理和をとる
E009    E2 E00D     JPO   E00D   ←— １の数が奇ならジャンプ
E00C    77          MOV   M,A    ←— 偶なら新しいデータをもとの番地へ
E00D    05          DCR   B      ←— カウンタ１減らす
E00E    C2 E006     JNZ   E006   ←— データがまだあればＥ００６へ
E011    76          HLT
```

データの個数は＄Ｅ０２０番地に入っており、データは＄Ｅ０２１番地から続いて入っています。パリティをつけたデータは＄Ｅ０２１番地からしまうことにしました。

プログラムを見ていきましょう。第１行目でポインタの値をＥ０２０に指定しています。第２行目の　ＭＯＶ　Ｂ，Ｍ　で＄Ｅ０２０番地の中味をＢレジスタに代入しています。Ｂレジスタがデータの個数のカウンタとして使われています。第３行目の　ＭＶＩ　Ｃ，８０　でＣレジスタに８０が入ります。16進法での８０の２進法での表現は１０

０００００００ですから、第８ビットを１にします。Ｃレジスタの第８ビットは１になっているわけです。第４行目でポインタの値を１だけすすめ、第５行目の　MOV A, M　でアキュムレータＡにデータをとりこみます。第６行目の　ＯＲＡ Ｃ　でアキュムレータＡの値とＣレジスタの値とのＯＲ演算をします。ＯＲ演算はどちらかが１であれば、結果は１になるというものです。右の表をごらんください。ＯＲＡ　Ｃは第８ビットを必ず１にすることがわかります。

第７行目の　ＪＰＯ Ｅ００Ｄ　はJUMP IF PARITY ODDであり、パリティが奇の場合にはＥ００Ｄにとびます。パリティが奇でない場合には第８行目の　MOV M, A　でアキュムレータの中味をデータ領域に返し、入れかえます。つまり、データはすべて偶のパリティに統一されていくのです。第９行目の　ＤＣＲ Ｂ　でデータの個数を１減らしています。第10行目の　ＪＮＺ Ｅ００６　はデータの個数が０でない限り＄Ｅ００６番地へとび処理を続行します。データがつきた時は第11行目のＨＬＴへすすみ終わりです。

ORA
(OR REGISTER OR MEMORY WITH ACCUMULATOR)

X OR Yの演算

Y＼X	0	1
0	0	1
1	1	1

JPO
(JUMP IF PARITY ODD)

```
E020 0A 30 31 32 33 34 35 36 37 38 39  ←── 実行前
E020 0A 30 B1 B2 33 B4 35 36 B7 B8 39  ←── 実行後
```

プログラムの実行前後のメモリの様子をみると正しくパリティが偶になっている様子がわかります。ただし日本語の使えるパソコンでは偶のパリティを使うと間違いが起こります。それは前のＡＳＣＩＩコード表（ＪＩＳコード表といったほうが適切かも知れません）をごらんください。

プログラム実行前のメモリの様子

```
E020 0A 30 31 32 33 34 35 36 37 38 39      0123456789:
```

プログラム実行後のメモリの様子

```
E020 0A 30 B1 B2 33 B4 35 36 B7 B8 39      0ｱｲ3ｴ56ｷｸ9
```

つまりプログラムの実行前後のメモリの様子をもう一度見てみますと、16進数で打ち出してみると３１はＢ１に３２はＢ２などになっていますが、ＡＳＣＩＩコードで打ち出してみますと１がカタカナのアに化けたり、２がイに化けたりします。日本語の使えるＪＩＳコードでは第８ビットも使っているので、このようなことが起こるのです。この点よく注意する必要があります。

5·2　同じ文字列かどうかを調べる

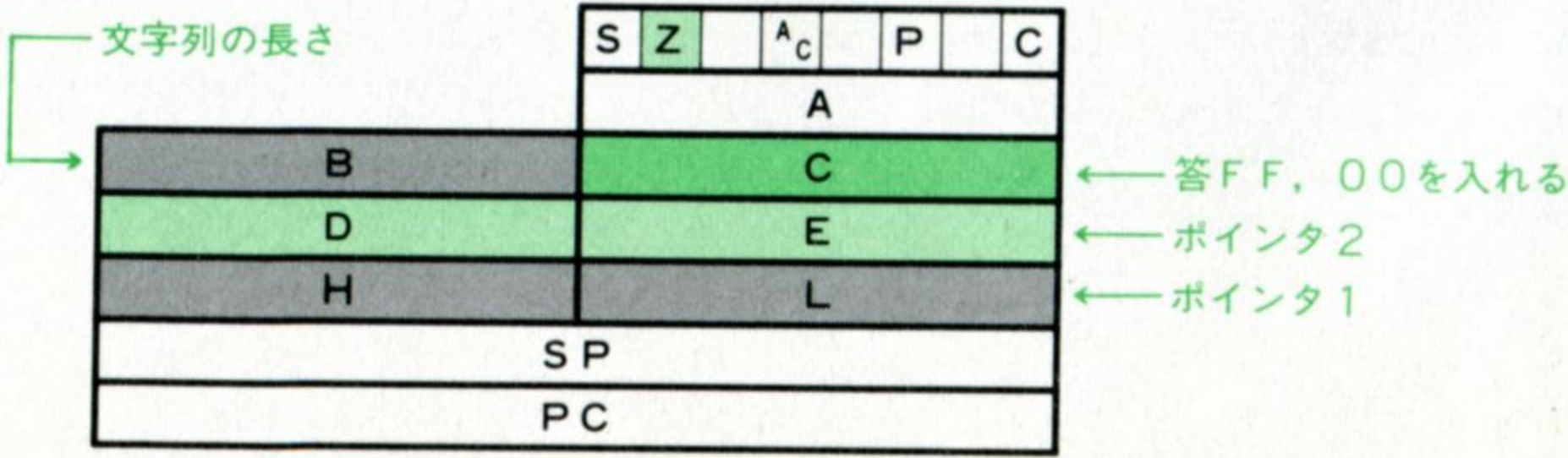
アドレス	内容
E000	メインプログラム
E01B	
	未使用
E020	答
E021	文字列の長さ
E022	文字列Ａ
E026	未使用
E032	文字列Ｂ
E036	未使用

ＩＮＴＬＥとＩＮＴＥＬは同じ文字列でしょうか。明らかに違いますね。しかし違うということはどういうふうにプログラムしたらよいでしょうか。それを考えるのがこのプログラムです。＄Ｅ０２２番地からはＩＮＴＬＥが、＄Ｅ０３２番地からはＩＮＴＥＬが入っています。これを比較し、同一であれば＄Ｅ０２０番地に００を、違っていれば＄Ｅ０２０番地にＦＦを代入することにします。

まず第１行目でポインタの値を＄Ｅ０２１番地にしています。第２行目でポインタの指し示す番地の中味をレジスタＢに代入しています。第３行目でポインタの値を１すすめています。第４行目でＤＥレジスタを第２のポインタとし、このポインタ２の値を＄Ｅ０３２番地にしています。第５行目でＣレジスタにＦＦを代入します。

```
E000   21 E021   LXI  H,E021   ← ポインタ1の設定
E003   46        MOV  B,M      ← 文字列の長さをレジスタへ
E004   23        INX  H        ← ポインタ1を1すすめる
E005   11 E030   LXI  D,E032   ← ポインタ2の設定
E008   0E FF     MVI  C,FF
E00A   1A        LDAX D        ← DEレジスタで示される番地の
                                  中味をアキュムレータへ
E00B   BE        CMP  M        ← 2数を比較
E00C   C2 E017   JNZ  E017     ← 違っていればジャンプ
E00F   13        INX  D        ┐
E010   23        INX  H        ┘ ポインタ1，2を1すすめる
E011   05        DCR  B        ← カウンタを1減らす
E012   C2 E00A   JNZ  E00A     ← 無条件ジャンプ
E015   0E 00     MVI  C,00     ← 同一の場合の処理
E017   79        MOV  A,C      ← 違う場合の処理
E018   32 E020   STA  E020     ← 結果を$E020番地へ格納
E01B   76        HLT
```

第６行目のＬＤＡＸ　ＤはLOAD ACCUMULATOR FROM MEMORY LOCATION ADDRESSED BY REGISTER

	S Z A~C~ P C	
	A	
B ← 文字列の長さ	C	← 答ＦＦ，００を入れる
D	E	← ポインタ２
H	L	← ポインタ１
SP		
PC		

ＰＡＩＲ でＤＥレジスタが指し示す番地の中味をアキュムレータＡに代入しています。第７行目の　ＣＭＰ Ｍ はアキュムレータの中味とＨＬレジスタ（ポインタ１）が指し示す番地の中味を比較します。

第８行目の　ＪＮＺ Ｅ０１７ は、もし結果が0でなければ＄Ｅ０１７番地へとび、ＭＯＶ Ａ，Ｃ でレジスタＣにしまわれているＦＦという答をアキュムレータＡにしまい、ＳＴＡ Ｅ０２０ でアキュムレータの中味を＄Ｅ０２０番地にしまって終了です。

結果が0であるときには第９行目の　ＩＮＸ Ｄ でポインタ２の値を１すすめ、第10行目の　ＩＮＸ Ｈ でポインタ１の値を１すすめます。

第11行目の　ＤＣＲ Ｂ は文字列の長さを１へらします。第12行目の　ＪＮＺ Ｅ００Ａ は文字列の長さが0になれば第13行目にすすみ、0になっていなければ＄Ｅ００Ａ番地からの処理をくり返します。

もしすべてが一致して正規に第13行目に入りますと、ＭＶＩ Ｃ，００ でレジスタＣに００を入れます。第14行目からの　ＭＯＶ Ａ，Ｃ でアキュムレータに００を入れ、第15行目の　ＳＴＡ Ｅ０２０ で＄Ｅ０２０番地に００を代入して、第16行目のＨＬＴで終わりです。

この例題ではＩＮＴＬＥとＩＮＴＥＬを比較していますので、答はＦＦで、プログラム実行後の＄Ｅ０２０番地を見てみますと正しくＦＦになっています。

プログラム実行前のメモリの様子

```
E020 00 05 49 4E 54 4C 45
E032 49 4E 54 45 4C
```

文字列Ａ（E020の49 4E 54 4C 45）
文字列Ｂ（E032の49 4E 54 45 4C）

プログラム実行後のメモリの様子

```
E020 FF 05 49 4E 54 4C 45
```

ＡとＢが違う文字列だったのでＦＦが入っている

5・3 16進数からＡＳＣＩＩ符号へ

16進数からＡＳＣＩＩ符号へ変換するプログラムを考えてみましょう。16進数の０はＡＳＣＩＩ符号では０ではありません。表からもわかるように３０です。これがなかなかわかりにくいところです。

そこで16進数からＡＳＣＩＩ符号の変換するには、

16進数＋＆Ｈ３０＝ＡＳＣＩＩ符号

としてやるとよいことがわかります。誤解があるといけないので一言注意しておきますが、ここでいう変換とは、16進数での０からＦがＡＳＣＩＩ符号で０からＦがでてくるような変換のことです。

	0	1	2	3	4	5	6	7
0	NUL	DLE		0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	¥	l	¦
D	CR	GS	−	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	

それでは最も簡単なプログラムを見ていきましょう。

CPI

JC
(JUMP IF CARRY)

第１行目の　ＬＤＡ Ｅ０２０ は＄Ｅ０２０番地の中味をアキュムレータに代入します。第２行目の　ＣＰＩ １０はCOMPARE ACCUMULATOR CONTENTS WITH IMMEDIATE DATAですから、アキュムレータＡの中味と０Ａすなわち10進数の１０とどちらが大きいかを見てみます。第３行目の　ＪＣ Ｅ００Ａ はアキュムレータの中味が０４より小さい場合には＄Ｅ００Ａ番地へすすみ、０Ａ以上の場合には第４行目の　ＡＤＩ ０７ へすすみます。

どうしてそんなことをするのかと疑問に思われる方は、もう一度ＡＳＣＩＩコードの表をみてください。０～９とＡ～Ｆの間にはとびが

```
E000   3A E020    LDA  E020
E003   FE 0A      CPI  0A
E005   DA E00A    JC   E00A
E008   C6 07      ADI  07
E00A   C6 30      ADI  30
E00C   32 E021    STA  E021
E00F   76         HLT
```

あります。これを処理するために場合わけが必要なのです。第４行目の　ADI　07　は、ASCIIコード表でのとび７を処理しています。第５行目の　ADI　30　はASCIIコードでの０の値３０を加えているのです。これをオフセットと呼びます。便利な言葉なので覚えておかれるとよいでしょう。第６行目では結果を＄E０２１番地にしまい、第７行目のHLTで終わりです。

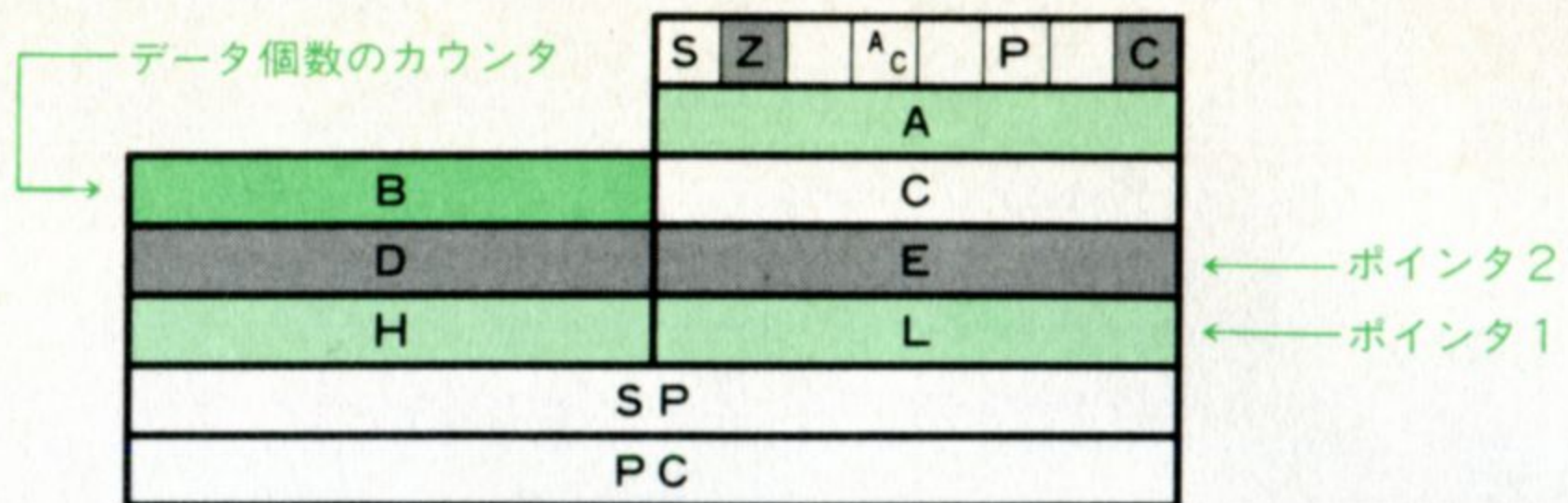
ADI
(ADD IMMEDIATE DATA WITHOUT CARRY)

オフセット

このままでもよいのですが、どうせいろいろ勉強して来たのですから、もう少しプログラムを改造して16進数の００から０Fを自動的にASCII符号へと変換するようなプログラムを作ってみましょう。下図に示すプログラムを見てください。第１行目ではポインタの値を＄E０２０番地にしています。これを第２行目の　MOV B, M　でレジスタBにしまいます。レジスタBはデータの個数のカウンタとして使われているのです。第３行目ではポインタの値を１だけすすめています。第４行目ではDEレジスタを第２番目のポインタに指定し、このポインタの値を＄E０４１番地にしています。第５行目ではアキュムレータAにポインタ１の指す＄E０２１番地のデータを格納しています。

番地	内容
E000–E019	メインプログラム
	未使用
E020	データの個数
E021–E031	16進数のデータ
	未使用
E041–E051	ASCIIデータ
	未使用

```
E000   21 E020    LXI  H,E020  ←ポインタ１の設定
E003   46         MOV  B,M     ←セットされた＄E０２０番地のデータ数をレジスタBへ
E004   23         INX  H
E005   11 E041    LXI  D,E041  ←ポインタ２の設定
E008   7E         MOV  A,M     ←レジスタMの中味をアキュムレータへ
E009   FE 0A      CPI  0A      ←16進のデータと０Aを比較
E00B   DA E010    JC   E010    ←キャリーがたてば，つまり16進のデータのほうが小さければジャンプ
E00E   C6 07      ADI  07
E010   C6 30      ADI  30
E012   12         STAX D       ←アキュムレータの中味をDEレジスタで示される番地へ格納
E013   13         INX  D       ┐
E014   23         INX  H       ┘←ポインタ１，２を１ずつすすめる
E015   05         DCR  B       ←カウンタを１減らす
E016   C2 E008    JNZ  E008
E019   76         HLT
```

データ個数のカウンタ →		S Z A_C P C	
	B	C	
	D	E	← ポインタ2
	H	L	← ポインタ1
	SP		
	PC		

STAX
(STORE ACCUMULATOR CONTENTS IN MEMORY AS ADDRESSED BY PEGISTER PAIR)

第６行目から第９行目までは前のプログラムと同じです。第10行目の　ＳＴＡＸ Ｄ でアキュムレータＡの中味をポインタ２のＤＥレジスタの指し示す番地に格納しています。第11行、第12行ではポインタ１とポインタ２の値を１ずつ進めています。つまり２本のポインタが並行して動いているといえるでしょう。第13行目の　ＤＣＲ Ｂ でデータの個数のカウンタを１減らしています。第14行目の　ＪＮＺ Ｅ００８ はデータが全部処理されたかどうかを見ています。順調にデータの処理が終わると第15行目のＨＬＴで終わりです。

このプログラムは一応動作はしますが、なお改良すべき余地があります。それはどこでしょうか。いろいろ考えてみてください。

プログラム実行前のメモリの様子

```
E020 10 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E
E030 0F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
E040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
E050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

プログラム実行後のメモリの様子

```
E020 10 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E
E030 0F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
E040 00 30 31 32 33 34 35 36 37 38 39 41 42 43 44 45
E050 46 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

5・4　ASCII符号から10進数へ

今度はASCII符号の０～９を10進数の０～９へ変換するプログラムを考えてみましょう。もし０～９以外のASCII符号がきた場合にはエラーとしてFFで表示することにします。下のプログラムを見てください。まず第１行目の　MVI　B，FF　でBレジスタにFFというエラーコードを入れています。第２行目の　LDA　E020　で、＄E020番地に入っているASCII符号をアキュムレータAに持ってきます。第３行目の　SUI　30　はSUBTRACT　IMMEDIATE　DATA　FROM　ACCUMULATORで、アキュムレータAの中味から16進数の30を引きます。第４行目の　JC　E0100　は、これまでたびたびでてきたものですが、アキュムレータAの中味が30より小さい場合にはキャリーフラグが１となるので＄E010番地へとびます。この場合、Bレジスタの中味はFFとなっていますから、第８行目の　MOV　A，B　でアキュムレータAの中味はFFとなります。第９行目の　STA　E021　でアキュムレータAの中味を＄E021番地へ格納して終わりです。

アキュムレータAの中味が30より大きい場合には第５行目のCPI　0A　にすすみます。すでに30をひいたものが、さらに10進数の10より小さいか小さくないかを調べます。つまり、ASCII符号の０～９の範囲に入っているかどうかを調べるのです。もし入っていればキャリーフラグは１ですからJNCにはひっかからず、第７行目の　MOV　B，A　で、アキュムレータAの中味をBレジスタにうつし弟８行目以下にすすみます。この場合はBレジスタの中味がエラーコードFFでなく、10進数になっていることだけが違います。

もしASCII符号の０～９の範囲に入っていない場合には、第６行目のJNCでキャリーフラグが０ですので、第８行目からのエラー処理へとすすみます。JCとJNCを使っていますので頭がゴチャゴ

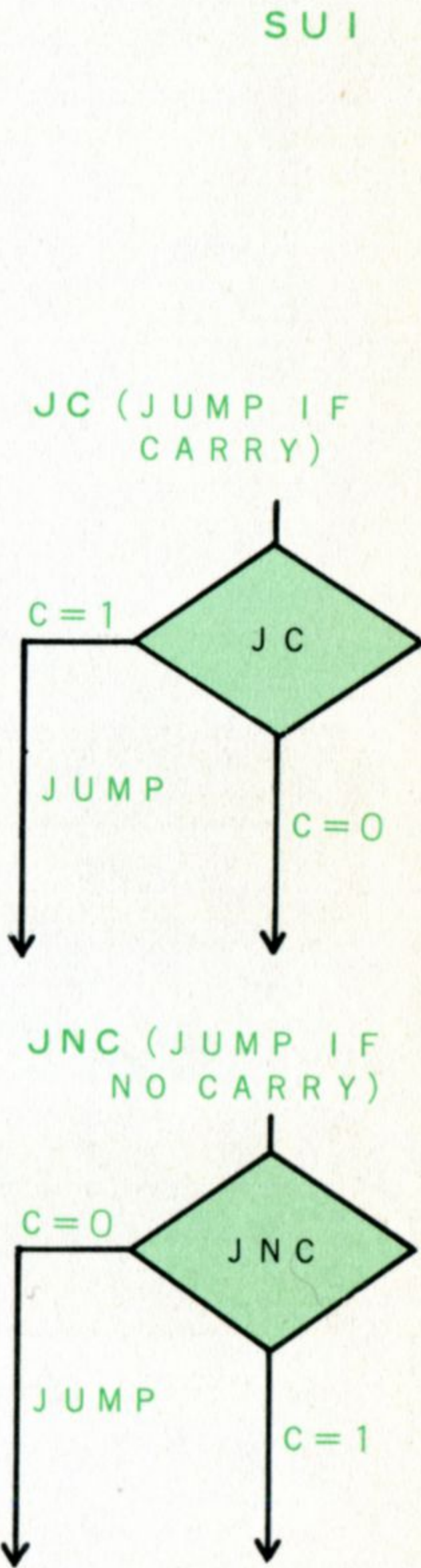
```
E000   06 FF      MVI  B,FF
E002   3A E020    LDA  E020
E005   D6 30      SUI  30
E007   DA E010    JC   E010
E00A   FE 0A      CPI  0A
E00C   D2 E010    JNC  E010
E00F   47         MOV  B,A
E010   78         MOV  A,B
E011   32 E021    STA  E021
E014   76         HLT
```

SUI

JC（JUMP IF CARRY）

C＝1　JC　JUMP　C＝0

JNC（JUMP IF NO CARRY）

C＝0　JNC　JUMP　C＝1

```
E000   21 E020   LXI  H,E020  ← ポインタ1
E003   46        MOV  B,M     ← データ数をレジスタBへ
E004   23        INX  H
E005   11 E030   LXI  D,E030  ← ポインタ2の設定
E008   0E FF     MVI  C,FF    ← エラーコードをレジスタCへ
E00A   7E        MOV  A,M     ← データをアキュムレータへ
E00B   D6 30     SUI  30      ← アキュムレータの中味から30を引き
                                 その値をアキュムレータへ格納
E00D   DA E016   JC   E016
E010   FE 0A     CPI  0A      ← アキュムレータの中味と10を比較
E012   D2 E016   JNC  E016    ← アキュムレータの中味のほうが
                                 大きければジャンプ
E015   4F        MOV  C,A
E016   79        MOV  A,C
E017   12        STAX D       ← ポインタ2で示される番地に
                                 アキュムレータの中味を格納
E018   13        INX  D
E019   23        INX  H
E01A   05        DCR  B       ← データ数を1減らす
E01B   C2 E008   JNZ  E008
E01E   76        HLT
```

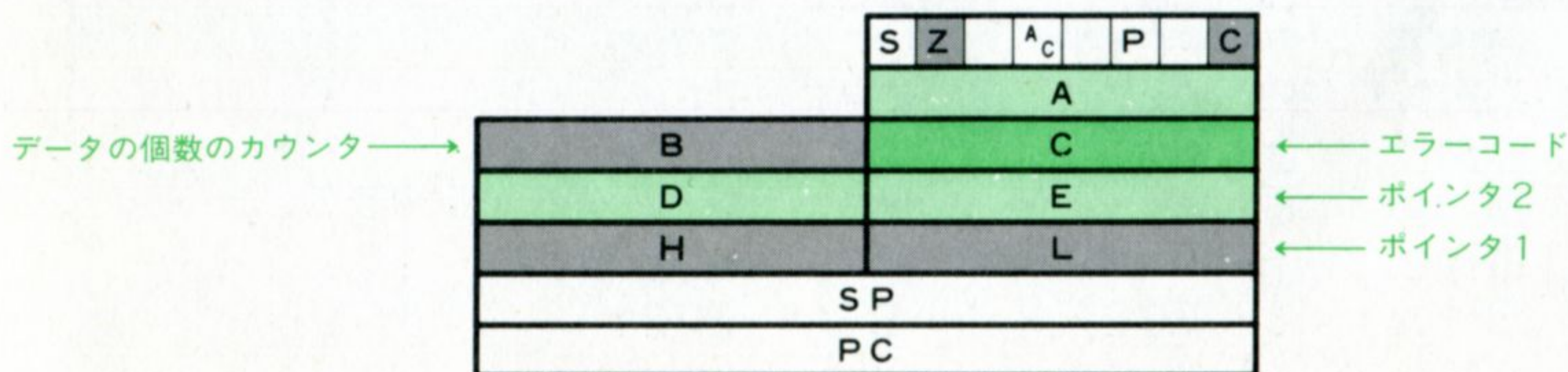
番地	内容
E000–E01E	メインプログラム
	未使用
E020	データの個数
E021–E02A	ASCIIデータ
	未使用
E030–E039	10進変換データ
	未使用

チャするかも知れませんが、欄外の図を見てください。

これでプログラムはできたのですが、5・3節と同じようにプログラムを改造してやることにしましょう。そのほうが楽に結果を見ることができます。プログラムを上に示します。

プログラムの考え方はこれまでの節とまったく同じですので、ここでは別の説明の仕方をしましょう。HLレジスタはポインタ1としてデータの個数とASCIIデータの格納場所を指し示すために用いられています。DEレジスタはポインタ2として10進変換データの格納場所を指し示すために用いられています。Bレジスタはデータの個数のカウンタとして使われています。Cレジスタはエラーコードを格納するために使われています。

複雑なプログラムを作るには、もっとたくさんレジスタがあって、もっとポインタを使えたら便利だと思うでしょう。そして8ビットでなくて16ビットのレジスタがもっとあったらと思いませんか。そう思

	S Z AC P C	
	A	
データの個数のカウンタ → B	C	← エラーコード
D	E	← ポインタ2
H	L	← ポインタ1
SP		
PC		

うようになってきたら相当の力がついてきたのです。実は８ビットマイクロプロセッサより一段と高度なアキュムレータをもつ16ビットマイクロプロセッサというものがあります。80系ではインテル８０８６／８０８８が有名です。しかし先をあせらないでください。16ビットは強力には違いありませんが、それなりになかなか難しいのです。まず８ビットで充分力をつけてから先へすすんでください。勉強したことは決して無駄にはなりません。

プログラム実行前のメモリの様子

```
E020 0A 30 31 32 33 34 45 46 47 48 49
E030 00 00 00 00 00 00 00 00 00 00 00
```

プログラム実行後のメモリの様子

データ数（E020 0A）　データがセットされている（30〜49）

```
E020 0A 30 31 32 33 34 45 46 47 48 49
E030 00 01 02 03 04 FF FF FF FF FF 00
```

10進変換データ（00〜04）　エラーコード（FF〜FF）

同じような動作をするＢＡＳＩＣプログラムを示しておきます。

```
100 FOR I=1 TO 16
110 R$="NOT DECIMAL NUMBER"
120 READ A
130 N=A-&H30
140 IF N<0 THEN GOTO 170
150 IF N>=10 THEN GOTO 170
160 R$="DECIMAL NUMBER IS "+CHR$(A)
170 PRINT R$
180 R$=""
190 NEXT I
200 END
210 '
220 DATA &H30,&H31,&H32,&H33,&H34,&H35,&H36,&H37
230 DATA &H38,&H39,&H3A,&H3B,&h3C,&H3D,&H3E,&H3F
```

5・5　ASCII符号列から2進数へ

表題から別のことを考えられる方がおられるかも知れません。たとえばＡＳＣＩＩ符号の３の16進数表現は33ですから、これを２進数で表現しますと００１１００１１になります。これを頭に描かれると困るのです。本節で扱う例題は16進数の列

30，30，31，31，30，30，31，31

はＡＳＣＩＩ符号では００１１００１１をあらわしているので、これを２進数としてみると33であるということなのです。少し頭がおかしくなりそうですね。

それでは一緒にプログラムを見ていきましょう。まず第１行目ではポインタの値を＄Ｅ０３２番地に指定しています。第２行目の ＳＵＢ Ａ は、アキュムレータＡの中味からアキュムレータＡの中味を引いていますから、アキュムレータを０にリセットすることになります。少々気どった人は ＸＯＲ Ａ などとしてもかまいません。第３行目の ＭＶＩ Ｂ，０８ はＢレジスタに８を代入しています。Ｂレジスタはデータの個数のカウンタとして使われていることがわかります。第４行目の ＭＶＩ Ｃ，ＦＦ はＣレジスタにＦＦを代入しています。これはＣレジスタにエラーコードを入れているのです。

アドレス	内容
E000	メインプログラム
E020	未使用
E030	2進変換データ
E031	未使用
E032	ASCII符号列 "30"，"31"
E039	未使用

```
E000    21 E032     LXI   H,E032  ←ポインタの設定
E003    97          SUB   A       ←アキュムレータをゼロクリア
E004    06 08       MVI   B,08    ←データ数をレジスタＢへ
E006    0E FF       MVI   C,FF    ←エラーコードをレジスタＣへ
E008    87          ADD   A       ←アキュムレータの中味を１ビット左へ
E009    57          MOV   D,A     ←結果をＤレジスタに待避させる
E00A    7E          MOV   A,M
E00B    D6 30       SUI   30      ←アキュムレータの中味から３０を引く
E00D    FE 02       CPI   02      ←ＡＳＣＩＩコードの０か１かのチェック
E00F    D2 E01A     JNC   E01A    ←０か１でなければジャンプ
E012    82          ADD   D       ←アキュムレータの中味とレジスタ
                                    Ｄの中味を加える
E013    23          INX   H
E014    05          DCR   B       ←カウンタを１減ずる
E015    C2 E008     JNZ   E008    ←データがあればジャンプ
E018    0E 00       MVI   C,00    ←Ｃレジスタに終了マークを
E01A    21 E030     LXI   H,E030  ←ポインタの設定
E01D    77          MOV   M,A     ←＄Ｅ０３０番地に結果を格納
E01E    23          INX   H
E01F    71          MOV   M,C     ←＄Ｅ０３１番地に終了マークを格納
E020    76          HLT
```

第５行目の　ＡＤＤ Ａ はアキュムレータの中味を２倍しますから、左へ１ビットシフトさせることになります。第６行目では左へ１ビットシフトさせた中味をＤレジスタに待避させています。第７行目のＭＯＶ Ａ，Ｍ ではポインタの指し示している番地の中味をアキュムレータＡに持ってきます。

第８行目の　ＳＵＩ ３０ はＡＳＣＩＩコードを16進数に変換しようとしているのです。

第９行目の　ＣＰＩ ０２ は加工されたデータがＡＳＣＩＩコードの０か１であるかを判定するものです。２を引いて正であれば０でも１でもありません。この時キャリーフラグは０ですから、第10行目のＪＮＣ Ｅ０１Ａ で16行目の ＬＸＩ Ｈ,Ｅ０３０ にとびます。第16行目ではポインタの値を＄Ｅ０３０番地に変更し、第17行目の ＭＯＶ Ｍ，Ａ でアキュムレータの中味を＄Ｅ０３０番地にしまいます。

つまり、エラーをおこしたデータを＄Ｅ０３０番地にしまうのです。第18行目でポインタの値を１すすめ、第19行目の　ＭＯＶ Ｍ,Ｃ　で＄Ｅ０３１番地にエラーコードＦＦをしまいます。

さて第10行目の判定で正しいデータだったらどうなるでしょうか。第11行目の　ＡＤＤ Ｄ でアキュムレータＡの中味とＤレジスタの中味が加えられます。始めはＤレジスタは０で、今アキュムレータＡには最下位ビットにだけ１か０が入っていますから、結果は最下位ビットに１か０が入るだけです。

第12行目でポインタの値が＄Ｅ０３３番地にすすみます。第13行目の　ＤＣＲ Ｂ ではデータの個数のカウンタが１だけ減少します。第14行目の　ＪＮＺ Ｅ００８ はデータがつきない限り＄Ｅ００８番地からのループを回りなさいということです。

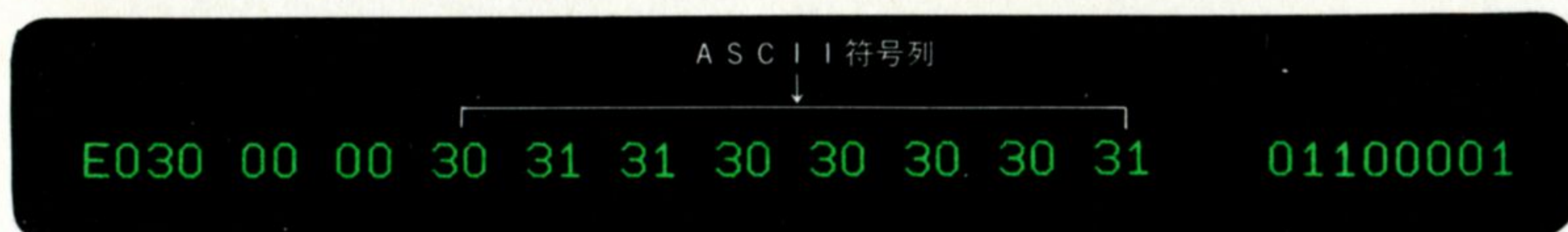
もう１回だけループを回ってみましょう。第５行目の　ＡＤＤ Ａ で左へ１ビットシフトします。第６行目でその結果をＤレジスタに待避させ、第７行目以降から、また次のデータをとり込むのです。第11行目の　ＡＤＤ Ｄ でＤレジスタに待避させておいたデータとアキュムレータの中味を加えています。こうして１つずつＡＳＣＩＩデータが２進データとしてとり込まれて２進数に変換されていくわけです。

全部のデータの処理が終わりますと第14行目の　ＪＮＺ Ｅ００８ にはひっかからなくなり、第15行目の　ＭＶＩ Ｃ，００ にすすみます。００は正常終了の印です。第16，17行目で最終結果が＄Ｅ０３０番地にしまわれ、第18，19行目でＣレジスタの正常終了のマークが＄Ｅ０３１番地にしまわれるわけです。

プログラム実行前のメモリの様子

```
                ASCII符号列
                     ↓
            ┌──────────────────────────────┐
E030 00 00 30 31 31 30 30 30 30 31      01100001
```

プログラム実行後のメモリの様子

```
E030 61 00 30 31 31 30 30 30 30 31 a 01100001
     ↑  ↑
     │  └────────── 正常終了のマーク
     └───────────── 2進数の01100001が入っている
```

同じような動作をするＢＡＳＩＣプログラムを示しておきます。

```
100 R$=""
110 E$="ERROR"
120 FOR I=1 TO 8
130 READ A
140 N=A-&H30
150 IF N<0 THEN GOTO 180
160 IF N>2 THEN GOTO 180
170 R$=R$+CHR$(A)
180 NEXT I
190 PRINT "BINARY NUMBER IS  ";R$:GOTO 210
200 PRINT E$
210 END
220 '
230 DATA &H30,&H31,&H31,&H30,&H30,&H30,&H30,&H31
```

第6章

やさしい計算こそむずかしい

6・1　精度の高い足し算

E000	メインプログラム
E015	未使用
E020	バイト数
E021	●加えられる数1 ●和
	未使用
E031	加えられる数2
	未使用

ANA
(AND REGISTER OR MEMORY WITH ACCUMULATOR)

LDAX
(LOAD ACCUMULATOR FROM MEMORY LOCATION ADDRESSED BY REGISTER PAIR)

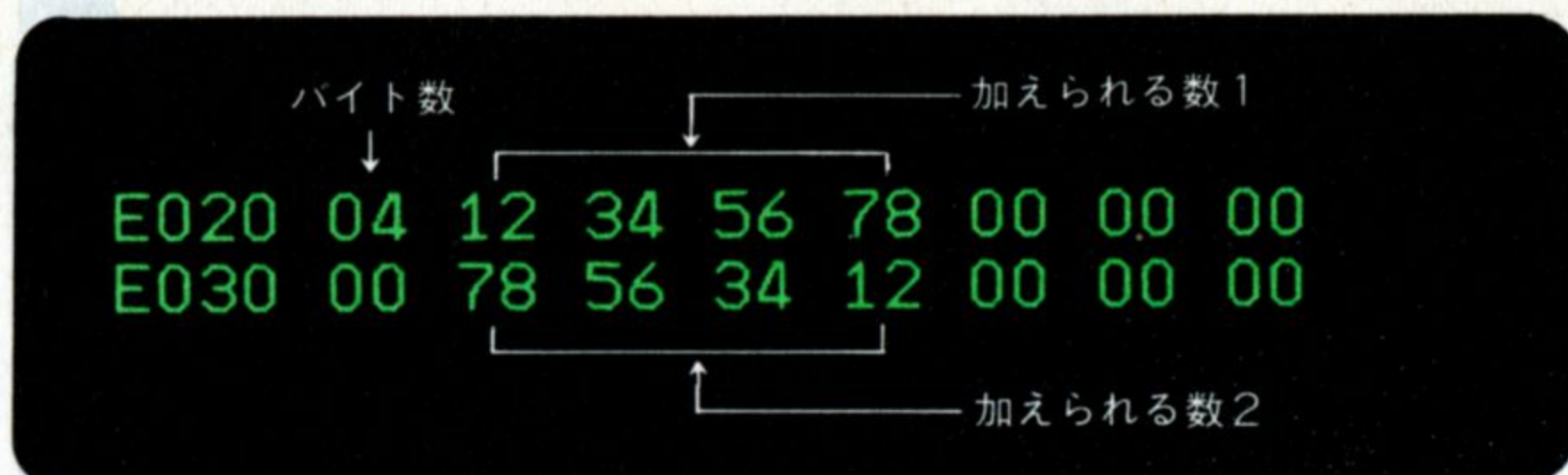
やさしい足し算については、すでに8ビットの足し算、16ビットの足し算を勉強していますが、32ビット、64ビットになったらどうしたらよいでしょうか。それを勉強するのが本節の目的です。

まずプログラムをみてください。第1行目ではアキュムレータAに＄E020番地の何バイトの計算であるかのバイト数をロードしています。第2行目ではこのバイト数をアキュムレータAからBレジスタに移します。第3行目、第4行目はHLレジスタをポインタ1にして＄E021番地に初期化し、DEレジスタをポインタ2にして＄E031番地に初期化しています。

第5行目の　ANA A　はアキュムレータAを0にリセットしています。もう1つCarry（キャリー：桁上がり）フラグを0にするという効果があります。ある命令がフラグをどう変更するかは、こまめに表を見て覚えるほかありません。だから、良いマイクロプロセッサを選ぶことが大切なのです。1人の人間にそんなにたくさん覚えることを望むのは残酷です。

第6行目の　LDAX D　はポインタ2のレジスタ対DEレジスタが指し示している番地の中味をアキュムレータAに持ってきます。第7行目の　ADC M　はポインタ1のレジスタ対HLレジスタが指し示している番地の中味をアキュムレータAにキャリーフラグと共に加えなさいということです。

```
E000    3A E020     LDA  E020
E003    47          MOV  B,A
E004    21 E021     LXI  H,E021  ←ポインタ1の設定
E007    11 E031     LXI  D,E031  ←ポインタ2の設定
E00A    A7          ANA  A       ←アキュムレータをリセット
E00B    1A          LDAX D       ←加えられる数2をアキュムレータへ
E00C    8E          ADC  M       ←2数をキャリーを含めて加算
E00D    27          DAA          ←2進化10進補正
E00E    77          MOV  M,A     ←補正値をポインタ1の示す番地へ格納
E00F    13          INX  D
E010    23          INX  H
E011    05          DCR  B       ←カウンタを1減らす
E012    C2 E00B     JNZ  E00B
E015    76          HLT
```

実行前のメモリの様子

```
    バイト数              加えられる数1
E020 04 12 34 56 78 00 00 00
E030 00 78 56 34 12 00 00 00
                      加えられる数2
```

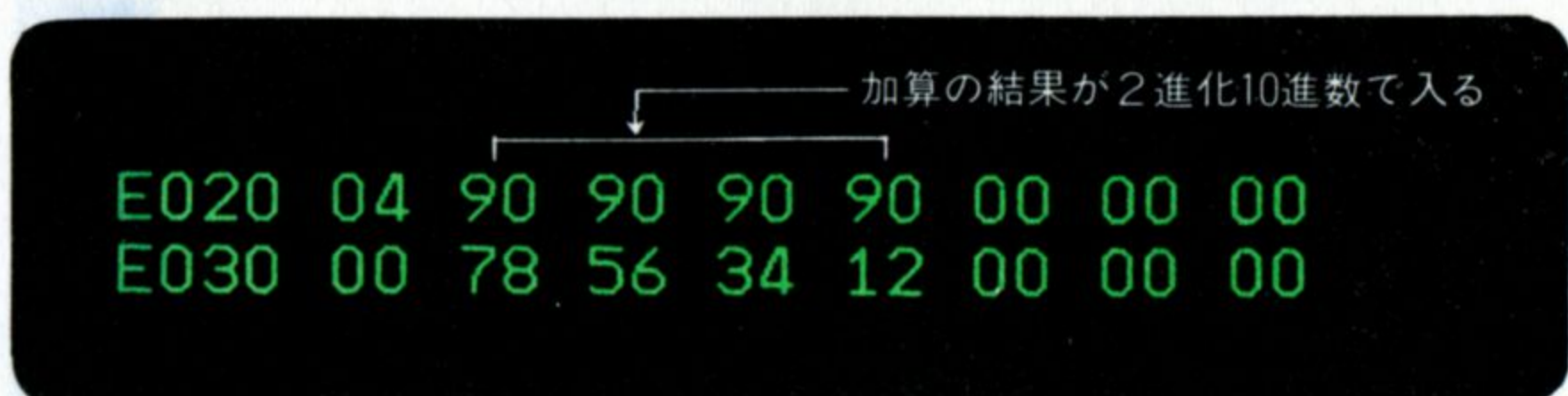
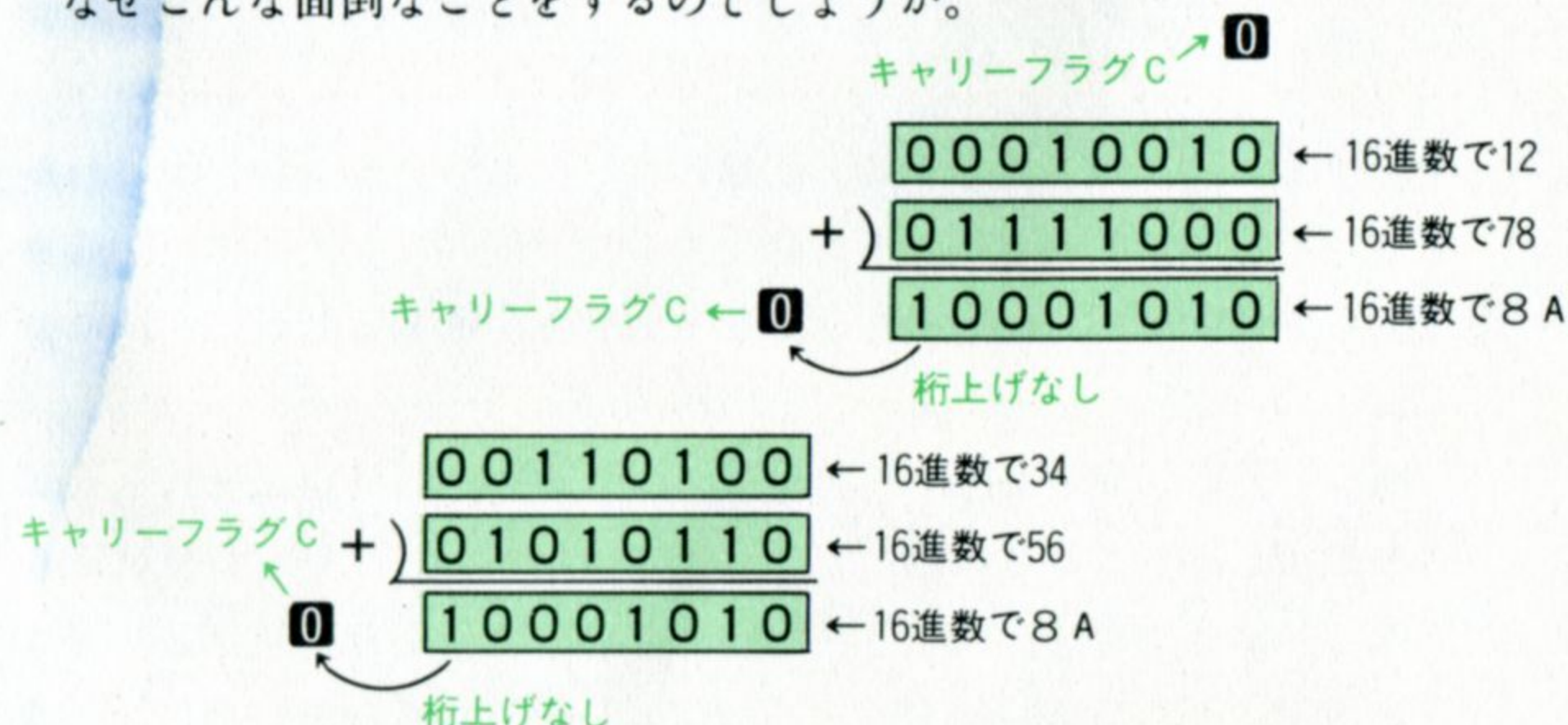
実行後のメモリの様子

```
                加算の結果が2進化10進数で入る
E020 04 90 90 90 90 00 00 00
E030 00 78 56 34 12 00 00 00
```

ADC
(ADD REGISTER OR MEMORY WITH CARRY TO ACCUMULATOR)

なぜこんな面倒なことをするのでしょうか。

```
                 キャリーフラグC ↗ 0
                    00010010  ←16進数で12
                 +) 01111000  ←16進数で78
キャリーフラグC ← 0  10001010  ←16進数で8A
                 桁上げなし

      00110100  ←16進数で34
キャリーフラグC +) 01010110  ←16進数で56
 0    10001010  ←16進数で8A
      桁上げなし
```

図をみるとその理由がわかりますね。実は次の計算をしているのです。

```
     78563412
  +) 12345678
```

という計算をしているのですが、一挙にはできないので、4回に分割して計算をしているのです。

第8行目のDAAはDECIMAL ADJUST ACCUMULATORでアキュムレータの中味をBCD (Binary Coded Decimal)コードすなわち2進化10進コードに変換しているのです。BCDコードは少し前の計測器には必ず採用されていたコードです。これは表示器に利用するのに便利だったためよく使われました。BCDコードの表を次にかかげておきましょう。最近はあまり使われない傾向にありま

DAA
(DECIMAL ADJUST ACCUMULATOR)

BCD
(BINARY CODED DECIMAL)

10進数	BCDコード		ASCIIコード	
0	00000000	00	00000000	00
1	00000001	01	00000001	01
2	00000010	02	00000010	02
3	00000011	03	00000011	03
〜	〜	〜	〜	〜
9	00010000	09	00001001	09
10	00010001	10	00001010	0A
11	00010010	11	00001011	0B
12	00010100	12	00001100	0C
13	00010101	13	00001101	0D
14	00010110	14	00001110	0E
15	00010101	15	00001111	0F
16	00010110	16	00010000	10
	2進表示	16進表示	2進表示	16進表示

す。つまり16進数で表示すると10進数でいくらであるかが一瞬にわかるというものです。電卓などに利用するには便利なものですが、時々混乱して何がなんだかわからなくなるような失敗をした覚えがあります。

第9行目の　MOV M, A　はアキュムレータAの中味をHLレジスタが指し示している番地にしまいます。第10行目、第11行目はポインタ1、ポインタ2の値を1ずつ進めています。第12行目はBレジスタの値を1へらしています。第13行目はBレジスタの値が0になったかどうか判定し、0でなければ$E00B番地からの処理をくり返します。第14行目で終わりです。

BCD補正をしない場合のプログラムは次ページの通りです。DAAをとっただけですから、特別な説明はしないことにします。

```
E000    3A E020      LDA  E020
E003    47           MOV  B,A
E004    21 E021      LXI  H,E021
E007    11 E031      LXI  D,E031
E00A    A7           ANA  A
E00B    1A           LDAX D
E00C    8E           ADC  M
E00D    77           MOV  M,A
E00E    13           INX  D
E00F    23           INX  H
E010    05           DCR  B
E011    C2 E00B      JNZ  E00B
E014    76           HLT
```

プログラム実行前のメモリの様子

加えられる数1

```
E020 04 12 34 56 78 00 00 00
E030 00 78 56 34 12 00 00 00
```

加えられる数2

プログラム実行後のメモリの様子

結果が16進数で入っている

```
E020 04 8A 8A 8A 8A 00 00 00
E030 00 78 56 34 12 00 00 00
```

6・2　8ビットのかけ算

	0	1
0	0	0
1	0	1

アドレス	内容
E000	メインプログラム
E01A	
	未使用
E020	被乗数
E021	乗数
E022	答・下位
E023	答・上位
	未使用

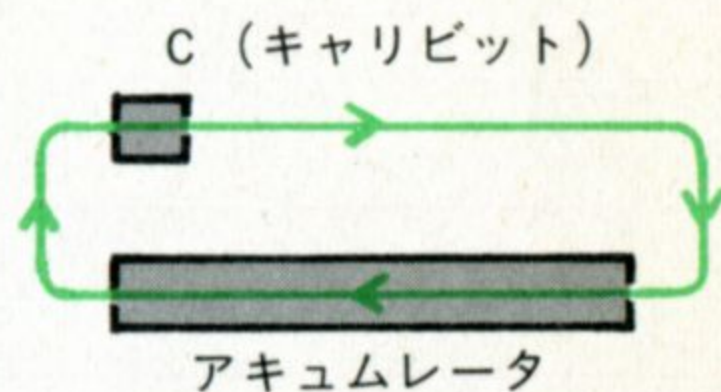
マイクロコンピュータが計算機であるなら、少なくとも加減乗除すなわち足し算、引き算、かけ算、わり算は自由にできそうなものですが、足し算、引き算は別として、かけ算、わり算はきわめて苦手です。かけ算も限られた範囲の整数の範囲なら楽なのですが、実数の範囲全体となると、浮動小数点計算用のパッケージが必要になってきます。これは相当難しくプロでないと作ることができません。我々はそこまで欲張ることなく、８ビットの整数同士のかけ算を勉強することにしましょう。我々は機械語でプログラムを組む以上、２進数のかけ算を扱うことになります。かけ算の九九に当たる表は左図のようになります。この表を機械的に適用するのも一つの方法ですが、かける数が0であると答は0だし、かける数が1だと、かけられる数を加えるだけだということを利用するとプログラムが作りやすくなります。

さてプログラムを見ていきましょう。第1行目でポインタの値を$Ｅ０２０番地に指定しています。第2行目でＥレジスタにかけられる数をしまっています。第3行目でＤレジスタに0を入れて16ビットの数に直しています。第4，５行目でポインタの値をすすめ、かける数をアキュムレータＡにしまっています。第6行目でＨＬレジスタに０を入れています。第7行目でＢレジスタに８を入れています。Ｂレジスタはビット数のカウンタに使われます。第8行目で　ＤＡＤ Ｈ はＨＬレジスタの中味にＨＬレジスタの中味を加えます。すなわちＨＬレジスタの中味が１ビット左へシフトすることになります。第９行目

```
E000    21 E020     LXI  H,E020
E003    5E          MOV  E,M    ←かけられる数をEレジスタに
E004    16 00       MVI  D,00   ←かけられる数を16ビットに直す
E006    23          INX  H
E007    7E          MOV  A,M    ←かける数をアキュムレータに
E008    21 0000     LXI  H,0000
E00B    06 08       MVI  B,08   ←カウンタをセット
E00D    29          DAD  H      ←データがまだあればHLレジスタの中味を
                                  2倍つまり1ビット左へずらす
E00E    17          RAL         ←かける数を1ビット左へ
E00F    D2 E013     JNC  E013
E012    19          DAD  D      ←キャリーがたっていれば現在の積に
                                  かけられる数を加える
E013    05          DCR  B
E014    C2 E00D     JNZ  E00D   ←カウンタが0でなければジャンプ
E017    22 E022     SHLD E022   ←結果を格納
E01A    76          HLT
```

のＲＡＬはROTATE ACCUMULATOR LEFT THROUGH CARRYで、アキュムレータＡの中味を左へ１ビットすすめます。この操作によってアキュムレータの中味が１ビットずつキャリービットＣへ送り込まれていきます。第10行目の　ＪＮＣ Ｅ０１３ は、もしかけられる数のキャリーが０であると何もしないで第12行目でＢレジスタの値を１へらし、キャリーが１であると第11行目でＨＬレジスタの中味（これは積に相当します）にＤＥレジスタの中味（これはかけられる数に相当します）を加えています。処理が終わると第12行目でＢレジスタの値を１へらします。いずれにせよＢレジスタの値を１へらして第13行目の　ＪＮＺ Ｅ００Ｄ で８ビット全部について、かけ算が終わったかどうか見ています。途中ならＥ００Ｄからのループを繰り返します。全部終了していれば第14行目の　ＳＨＬＤ Ｅ０２２ で積を＄Ｅ０２２番地、＄Ｅ０２３番地に入れて終わりです。＄Ｅ０２２番地に答の下位、＄Ｅ０２３番地に答の上位が入っていることに注意してください。

RAL
(ROTATE ACCUMULATOR LEFT THROUGH CARRY)

C（キャリビット）
アキュムレータ

この計算の仕組はきわめてわかりにくいので、図解してお見せしましょう。簡単のため、＄Ｅ０２０番地、＄Ｅ０２１番地には次の数が入っているものとします。

＄Ｅ０２０番地　　０５（かけられる数）
＄Ｅ０２１番地　　０３（かける数）

キャリーがたったのでＤＡＤ Ｄで、現在の積に加算され、カウンタが０でないのでもう一度ループを回り、ＤＡＤ Ｈで桁をずらす。つまり０５の２進法表現０００００１０１が１ビット左へずれて０００００１０１０（０Ａ）となる。

０Ａに０５を加算してループから抜ける。

	始め								終わり
途中の積	0	0	0	0	0	0	0	05	0F
かける数	03	06	0C	18	30	60	C0	80	01
かけられる数	05	05	05	05	05	05	05	05	05
カウンタ	8	7	6	5	4	3	2	1	0
キャリー	*	0	0	0	0	0	0	1	1

10進２　３×５＝15
だから０Ｆでよい。

乗数３の２進表現０００００００１１が１ビット左へずれ最上位ビットがキャリーに入る

頭のほうからかけ算をしているという感じになります。普通のお尻のほうからやるのとは逆になっています。

6・3　8ビットのわり算

かけ算もそうであったように、ＰＣ-８８０１のマイクロプロセッサＺ80にはわり算の命令は準備されていませんので、何とか工夫してやらねばなりません。わり算は本質的に引き算の繰り返しであると考えることができます。たとえば９÷２は９から２を何回引けるかということになると考えられます。９から２を繰り返し引いていきますと４回引けて、余りが１となります。この考え方を使うことになります。

プログラムを見ていきましょう。第１行目でＨＬレジスタに＄Ｅ０２０番地、＄Ｅ０２１番地の中味を代入しています。つまりＨＬレジスタにわられる数が入ったわけです。次に第２行目でアキュムレータＡに＄Ｅ０２２番地の中味すなわち、わる方の数を代入しています。第３行目でアキュムレータＡの中味をＣレジスタに移しています。第４行目はＢレジスタにカウンタとして８を入れています。第５行目のＤＡＤ　Ｈ　はＨＬレジスタの中味にＨＬレジスタを加えることですから、ＨＬレジスタの中味は２倍されます。２倍することでＨＬレジスタの中味は左へ１ビットだけシフトされます。第６行目のＭＯＶ　Ａ，Ｈ　はＨレジスタの中味をアキュムレータＡにうつします。実はＨレジスタにはわられる数の上位、Ｌレジスタにはわられる数の下位が入っていますので、アキュムレータにはわられる数の上位が入ってきます。第７行目の　ＳＵＢ　Ｃ　で、わられる数の上位から、わる数を引いています。第８行目の　ＪＣ　Ｅ０１１はキャリー　フラグＣを見ています。キャリーフラグが１ならばわられる数の上位のほうがわる数よりも小さいわけですから、わり算は終了して＄Ｅ０１１番地にとん

アドレス	内容
Ｅ０００	メインプログラム
Ｅ０１８	未使用
Ｅ０２０	被除数　下位
Ｅ０２１	被除数　上位
Ｅ０２２	除数
Ｅ０２３	答
Ｅ０２４	余り
	未使用

```
E000   2A E020    LHLD  E020   ← 被除数をＨＬレジスタへ
E003   3A E022    LDA   E022
E006   4F         MOV   C,A    ← 除数をＣレジスタへ
E007   06 08      MVI   B,08   ← カウンタをセット
E009   29         DAD   H      ← ＨＬレジスタを１ビット左へシフト
E00A   7C         MOV   A,H    ← 被除数と上位桁をアキュムレータへ
E00B   91         SUB   C      ← Ａの中味から除数を引く
E00C   DA E011    JC    E011   ← 除数のほうが大きければジャンプ
E00F   67         MOV   H,A    ← 除数を引いた余りをＨレジスタへ
E010   2C         INR   L      ← 答えを１プラス
E011   05         DCR   B
E012   C2 E009    JNZ   E009
E015   22 E023    SHLD  E023   ← 答とと余りを格納
E018   76         HLT
```

●＆H３E４F÷＆H６５の場合

上位３EがHレジスタへ、下位４FがLレジスタへはいります。

また、わる数６５はCレジスタに格納されます。

3E　4F　65

00111110　01001111　01100101

Hレジスタ　Lレジスタ　Cレジスタ

DAD H

01111100　10011110

アキュムレータ

01111100

01111100
01100101
00010111

SUB C　キャリーは0

Hレジスタへ

00010111

アキュムレータ

00010111　10011110

INR L　1がたつ

●キャリーがたたなかったのでこの桁での演算では除数が１つあったことになる

00010111　10011111

ここでDCR　Bでカウンタは7になる

DHD H

00101111　00111110

アキュムレータ

00101111

SUB C

00101111　00111110

●キャリーがたつので＄E０１１番地へジャンプ。つまり、この桁での演算では除数は0個。

カウンタは6に

DAD H

01011110　01111100

●以上の過程をカウンタが0になるまで繰り返す

01011110　10011101

余り　答え

5E　9D

で、Bレジスタの中味を1だけへらします。Bレジスタの中味が0でなければ、わり算は終了していませんので、$E009番地からのループへとぶことになります。

第8行目でキャリーフラグが0ならばわられる数の上位がわる数の上位よりも大きいか等しい（これを小さくないといいます）ので9行目の　MOV H, A でアキュムレータの中味をHレジスタにうつします。このときアキュムレータの中味はわられる数の分だけひかれていますのでHレジスタにはひかれたわられる数が入っているのです。また10行目でLレジスタの値をふやしています。Lレジスタにはわり算の答が入っているのです。第9行目以降の動きは説明なしでわかるでしょう。

プログラム実行前のメモリの様子

被除数が下位、上位の順に入っている
除数

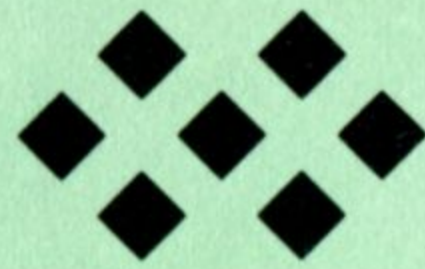
```
E020 4F 3E 65 00 00 00 00 00
```

プログラム実行後のメモリの様子

```
E020 4F 3E 65 9D 5E 00 00 00
```

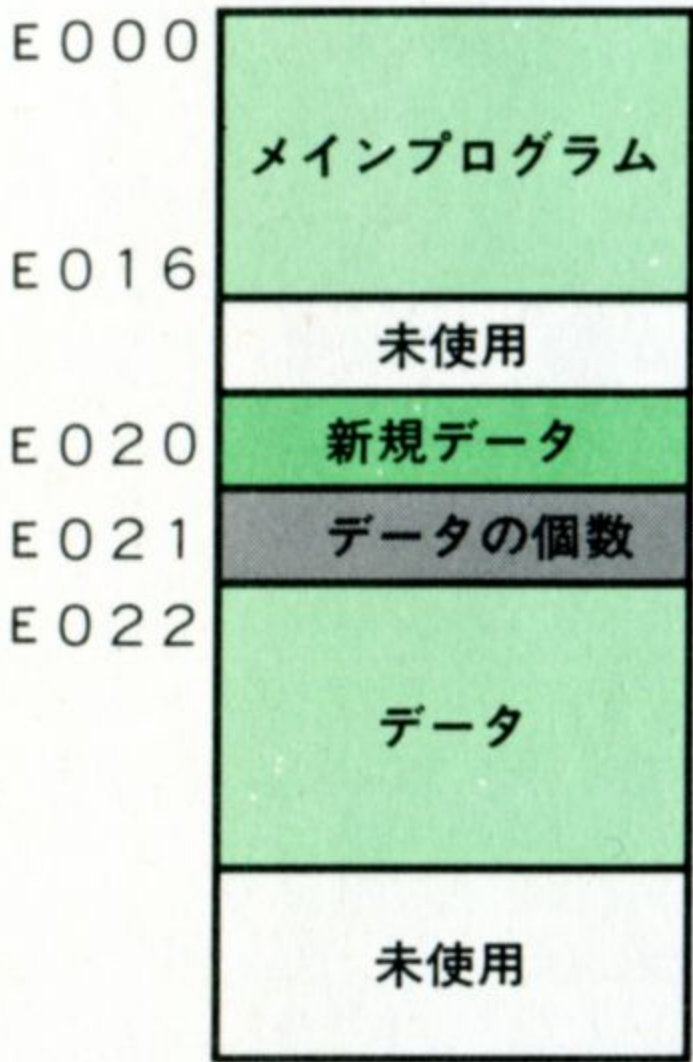
余り
答え

第7章

表とリストの使い方

7・1　新規データをリストに加える

このプログラムはあらかじめでき上っているデータのリストの中に新規のデータをつけ加えるものです。これを繰り返していくと、どんどんリストが自己増殖していくわけです。プログラムを見てみましょう。第１行目でＨＬレジスタに＄Ｅ０２１を代入しポインタの値を＄Ｅ０２１番地にしています。第２行目でＢレジスタにデータの個数を代入しています。第３行目でポインタの値を１つ進めています。第４行目でアキュムレータＡに＄Ｅ０２０番地の中味を持ってきます。すなわちアキュムレータＡに新規データが入ったわけです。第５行目のＣＭＰ　Ｍ　はアキュムレータの中味とリストの第１番目のデータとを比較します。第６行目ではもし一致していれば＄Ｅ０１６番地へとび終了です。リストの中に該当するデータがあったからです。もし一致していなければ第７行目にすすんで、ポインタの値を１すすめます。

第８行目ではデータの個数を１つ減らします。第９行目ではデータの個数が０でなければ＄Ｅ００８番地からの処理を繰り返しなさいということで、すべてのデータに当たって正常にループから脱出したときはリストの中に新しいデータはなかったわけですから、第10行目でアキュムレータＡの中味をリストの一番最後にしまいます。第11行目では、ポインタを＄Ｅ０２１番地にセットし、第12行目でポインタの指し示す番地の中味を１だけ増加します。それはリストのデータが１つ増えたからです。第13行目のＨＬＴで終了です。

アドレス	内容
E000	メインプログラム
E016	未使用
E020	新規データ
E021	データの個数
E022	データ
	未使用

```
E000    21 E021     LXI  H,E021
E003    46          MOV  B,M    ←カウンタをセット
E004    23          INX  H      ←ポインタを１すすめる
E005    3A E020     LDA  E020   ←新規データをアキュムレータへ
E008    BE          CMP  M      ←リスト中のデータと新規のデータを比較
E009    CA E016     JZ   E016
E00C    23          INX  H      ←ポインタを１すすめる
E00D    05          DCR  B      ←カウンタを１へらす
E00E    C2 E008     JNZ  E008   ←データがあればループ
E011    77          MOV  M,A    ←新規データをリスト末尾に格納
E012    21 E021     LXI  H,E021 ←ポインタのセット
E015    34          INR  M      ←新しいデータ数を格納
E016    76          HLT
```

プログラム実行前のメモリの様子

新規のデータ

E020 07 06 01 02 03 04 05 06 00

データ数　データリスト

プログラム実行後のメモリの様子

E020 07 07 01 02 03 04 05 06 07

追加された新規のデータ

応用として、ある漢字があらかじめ用意されている漢字のリストにあるかどうかを調べるプログラムを作ってみましょう。このプログラムでは漢字は255個までしかリストに登録できませんが、考え方だけは役に立つと思います。別々の会社のパソコンと漢字プリンタをつないだりする場合、欠字といって対応する漢字のないことがあるのです。そういう意味で実用的には大切な問題なのです。

```
E000  21 E032   LXI  H,E032
E003  46        MOV  B,M     ← カウンタのセット
E004  23        INX  H
E005  3A E030   LDA  E030    ← 漢字コードの下位をAへ
E008  BE        CMP  M       ← 2数を比較
E009  23        INX  H       ← ポインタを1すすめる
E00A  C2 E014   JNZ  E014    ← 一致しなければジャンプ
E00D  3A E031   LDA  E031    ← 下位が一致した場合、上位をAへ
E010  BE        CMP  M
E011  CA E026   JZ   E026    ← 一致すれば終わり
E014  23        INX  H       ← ポインタを次の漢字データへ
E015  05        DCR  B       ← カウンタを1へらす
E016  C2 E005   JNZ  E005    ← まだデータがあればループ
E019  11 E030   LXI  D,E030  ← ポインタをセット
E01C  1A        LDAX D       ← 漢字コードの下位をAへ
E01D  77        MOV  M,A     ← 上記の下位をデータリストの末尾へ
E01E  13        INX  D       ┐
E01F  23        INX  H       ┘← ポインタを1すすめる
E020  1A        LDAX D       ← 上位をアキュムレータへ
E021  77        MOV  M,A     ← 上記の上位をリスト末尾へ
E022  13        INX  D       ← ポインタを1すすめる
E023  1A        LDAX D       ← データ数をAへ
E024  3C        INR  A       ← データ数を1ふやす
E025  12        STAX D       ← 新しいデータ数をもとの場所へ
E026  76        HLT
```

（E019～E025：データリストになかった場合の処理）

E000	メインプログラム	
E026	未使用	
E030	調べる漢字のコード	下位
E031		上位
E032	データの個数	
E033	漢字データリスト	
E041	未使用	

さて漢字は２バイト・コードすなわち16ビットコードなので、前のプログラムを少し改造してやる必要があります。本質的に難しいところはありません。

プログラムが少し長いですし、１行１行説明するのは煩しいですから考え方だけをお話しすることにしましょう。まずBレジスタに漢字データの個数を入れてカウンタにしています。次に調べる漢字コードが漢字データのリストの中にあるかどうかを見てみます。具体的には調べる漢字コードの下位を先に比較しています。下位が一致しなければ、上位が一致しようとしまいと同一のコードではないのですから次の漢字コードへ進みます。下位が一致した場合は上位が一致するかどうかを調べます。一致した場合は、調べている漢字はもともとの漢字リストの中にあったことになりますので、それで終わりにします。一致しない場合は次々に全部の漢字を調べていきます。全部の漢字を調べても見つからない場合には、漢字データのリストに調べている漢字をつけ加えてやります。またデータの個数が１つふえたわけですから、データの個数を１だけふやします。

このプログラムの実行結果を見るためのＢＡＳＩＣプログラムを用意しました。漢字ボードが入っていることが前提になっていますが、短いプログラムなので簡単に入力できるでしょう。始めの方でお断りしておきましたようにＭＯＮＩＴＯＲプログラムを呼び出す前に

```
CLEAR ,&HDFFF
```

は実行しておいて頂けたでしょうか。これを実行していないとＢＡＳＩＣのプログラムが機械語のプログラムを破壊することがありますので必ず確認しておいてください。ＧＥ０００で機械語プログラムを走らせた後でＣＴＲＬ(コントロール)キーとＢのキーを同時に押してＢＡＳＩＣモードに戻して下さい。それからＢＡＳＩＣプログラムを入れて下さい。ＲＵＮとすれば114ページのような出力結果がでるはずです。

```
100 CLS 3
110 M=PEEK(&HE032)               ←データ数を読み取る
120 FOR I=1 TO M
130 N1=PEEK(&HE032+2*I-1)        ←漢字データの下位を読み取る
140 N2=PEEK(&HE032+2*I)          ←漢字データの上位を読み取る
150 N=N2*256+N1                  ←漢字コードの設定
160 PUT (I*20,20),KANJI(N),PSET,7,0  ←漢字を表示
170 NEXT I
180 END
```

このプログラムではいろいろ物足りない点があります。まず漢字データのリストの探索に時間がかかることです。もちろん7個ぐらいのデータでは大したことはありませんが、7千字位の漢字データのリストを調べることになると、もう少しうまいやり方を工夫したほうがよいのです。いまのプログラムでは漢字データが雑然と並んでいることを前提しているので、もし調べている漢字がリストにない場合には7千字全部を探索しなければならないのです。1秒でも1ミリ秒でも1マイクロ秒でも早くというプログラムの鉄則から考えると我慢できないことです。このためにはうまいやり方があります。それは漢字データのリストをコード順に並べておくことです。このテクニックについては次節で勉強することにします。

もう1つはBASICプログラムと機械語プログラムの結合のさせ方です。ここでのやり方はわかりやすいのですが、どうもスマートではありません。USR文とかCALL文というテクニックを使うともう少しスマートになります。しかし、それには準備と勉強が必要になるので、先へ進んでから勉強することにしましょう。

機械語プログラムを走らせる前にBASICプログラムを走らせた場合の画面表示は次のようになります。

```
亜唖阿哀愛挨
```

プログラムを走らせる前のメモリの状態

```
     調べる漢字コード              6個の漢字データ
E030 23 30 06 21 30 22 30 24 30 25 30 26 30 27 30 00
E040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

$E030，$E031番地に&H3023が入っており、漢字コード&H3023は娃という漢字に対応します。普通電算写植では欠字となっている難しく使われない漢字です。この漢字が漢字データのリストにあるかどうか調べようとしています。

機械語プログラムを走らせた後にＢＡＳＩＣプログラムを走らせた場合の画面表示は次のようになります。

亜唖阿哀愛挨娃

プログラムを走らせた後のメモリの状態

データ数が１ふえる

```
E030 23 30 07 21 30 22 30 24 30 25 30 26 30 27 30 23
E040 30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

調べる漢字コードがデータリストになかったので後ろに追加される

やはり娃という漢字は漢字データのリストには入っていなかったので、リストの一番最後につけ加えています。日本語ワープロと電算写植機を連動させるときには、電算写植機の漢字データのリストに欠字がないかどうか徹底的に調べます。もちろん、こんなやさしいプログラムではありませんが、気分だけは楽しめるでしょう。ＰＣ-８８０１でも本格的に使いこめばそうとうのことができます。

7・2　順番に並んだリストを調べる

今度はデータを大きさの順番に並べたリストに新規データがあるかどうかを調べることにしましょう。データが順番に並んでいるという保証があれば、ある新規データがリストの中にあるかどうかを調べるのはきわめて簡単になります。なぜなら、その数より大きくないか小さくないかのどちらかの数まで調べれば十分だからで全部のデータを調べる必要はないからです。それではプログラムを見ていきましょう。

第１行目、第２行目でＢレジスタにリストのデータの個数を入れています。第３行目でポインタの値を１だけ進めています。第４行目ではＣレジスタに０を入れています。第５行目でアキュムレータＡに＄Ｅ０２０番地の内容すなわち新規データを持ってきます。第６行目で新規データとリスト内のデータを比較しています。もし一致すれば＄Ｅ０１８番地へ飛んでアキュムレータＡにＣレジスタの中味０をしまい、第14行目の　ＳＴＡ　Ｅ０２Ｆでアキュムレータ A の中味を＄Ｅ０２Ｆ番地へしまいます。つまりリストの中に該当するものがあったということです。

第６行目でもし一致しなければ第７行目にすすみます。アキュムレータＡの中味よりリストの数のほうが大きいとキャリーＣが１になりますので　ＪＣ　Ｅ016で＄Ｅ０１６番地へとぶことになります。＄Ｅ０１６番地では　ＭＶＩＣ，ＦＦ　でＣレジスタにＦＦを入れて、最終的には＄Ｅ０２Ｆ番地にＦＦを格納します。つまりリストの中には該当するものがなかったということです。

```
E000    21 E021     LXI   H,E021
E003    46          MOV   B,M    ←カウンタのセット
E004    23          INX   H      ←ポインタをデータ領域へ移す
E005    0E 00       MVI   C,00   ←目的のデータがあったときの
                                   マークをCレジスタへ
E007    3A E020     LDA   E020
E00A    BE          CMP   M      ←２つのデータを比較
E00B    CA E018     JZ    E018   ←一致したときジャンプ
E00E    DA E016     JC    E016   ←該当するデータがないとき
E011    23          INX   H      ←ポインタを次のデータへ
E012    05          DCR   B      ←カウンタを１へらす
E013    C2 E00A     JNZ   E00A
E016    0E FF       MVI   C,FF
E018    79          MOV   A,C
E019    32 E02F     STA   E02F
E01C    76          HLT
```

アキュムレータAの中味よりリストの数のほうが大きくないときには第9行目にすすみ、ポインタの値を1だけすすめます。そして第10行目でBレジスタの値を1だけ減らします。第11行目はデータの個数が0になったかどうかを判定するもので、0になっていなければE00Aからのループを回ることになります。

プログラム実行前のメモリの様子

```
新規のデータ        データ
E020 07 05 01 02 03 04 05 00 00 00 00 00 00 00 00 00
        データ数
```

プログラム実行後のメモリの様子

```
E020 07 05 01 02 03 04 05 00 00 00 00 00 00 00 00 FF
                    該当するデータがなかったのでＦＦが入る
```

今度も漢字データのリストを調べるプログラムを作ってみましょう。調べようとしている漢字がリストの中にあるかどうかを見ることにします。プログラムの考え方はまず漢字の上位コードを見て、もし比較

```
E000    21 E032    LXI  H,E032
E003    46         MOV  B,M    ← カウンタのセット
E004    0E 00      MVI  C,00
E006    23         INX  H      ┐ ポインタを２つすすめて漢字デー
E007    23         INX  H      ┘ タの上位のところへ移す
E008    3A E031    LDA  E031   ← 調べる漢字コードの上位をＡへ
E00B    BE         CMP  M      ← ２数を比較
E00C    C2 E01E    JNZ  E01E   ← 一致しなければジャンプ
E00F    DA E022    JC   E022   ← 調べる範囲をこえていればジャンプ
E012    2B         DCX  H      ← ポインタを１へらして下位へ移す
E013    3A E030    LDA  E030   ← 調べる漢字コードの下位をＡへ
E016    BE         CMP  M      ← ２数を比較
E017    CA E024    JZ   E024   ← 一致したらジャンプ
E01A    DA E022    JC   E022   ← 調べる範囲をこえていればジャンプ
E01D    23         INX  H
E01E    05         DCR  B
E01F    C2 E006    JNZ  E006   ← データがあればループへ
E022    0E FF      MVI  C,FF   ← 見つからないときの処理
E024    79         MOV  A,C
E025    32 E04F    STA  E04F   ← ＄Ｅ０４Ｆ番地へＦＦか００を格納
E028    76         HLT
```

している漢字の上位コードのほうが大きければ、すでに調べるべき範囲を越えているわけですから、漢字データのリストの中にはなかったというしるしのＦＦを＄Ｅ０４Ｆ番地にしまって終わりにします。上位コードが大きくないときには、下位コードを比較します。こうしますとまた調べるべき範囲がせばめられるわけです。もし調べている漢字が漢字データのリストの中にあったときには＄Ｅ０４Ｆ番地に００をしまって終わりにしています。

漢字コードについてはＰＣ-８８０１のマニュアルを見てください。少し奇妙に感じられるかも知れないのはメモリに漢字コードが下位、上位の順に入っていることです。たとえば３０２３なら２３が先で、３０が後に入っています。実際の日本語ワードプロセッサでは空いている第８ビットをいろいろな機能の指定に使ったりしますのでＢ０Ａ３のように入ったりしています。こうした仕掛けやワナを一つ一つ見破っていくのはとても面白いことですが、本書の主題とは離れますので、この位にしておきましょう。

さて機械語プログラムの実行結果を見やすくするためにＢＡＳＩＣの助けを借りることにしましょう。ＰＣ-８８０１のＢＡＳＩＣは大変強力で頼りがいがあります。

ＢＡＳＩＣプログラムを簡単に説明しておきましょう。第１行目のＣＬＳ３は画面の字とグラフィックを消すものです。漢字はグラフィ

アドレス	内容
Ｅ０００	メインプログラム
Ｅ０２８	未使用
Ｅ０３０	調べる漢字コード
Ｅ０３１	
Ｅ０３２	データの個数
Ｅ０３３	漢字データリスト
	未使用
Ｅ０４Ｆ	答（ＦＦか００）
	未使用

```
100 CLS 3
110 S1=PEEK(&HE030) ←— 調べる漢字コードの下位を読む
120 S2=PEEK(&HE031) ←— 調べる漢字コードの上位を読む
130 S=S2*256+S1 ←— 漢字コードの設定
140 PUT (20,20),KANJI(S),PSET,7,0 ←— 漢字の画面表示
150 LOCATE 6,3
160 PRINT "ﾄ ｲｳ ｶﾝｼﾞ ﾊ ﾂｷﾞﾉ ﾘｽﾄ"
170 M=PEEK(&HE032) ←— データ数を読む
180 FOR I=1 TO M                        ┐
190 N1=PEEK(&HE032+2*I-1)              │
200 N2=PEEK(&HE032+2*I)                │
210 N=N2*256+N1                         ├— リスト中の漢字を画面に表示
220 PUT (I*20,60),KANJI(N),PSET,7,0    │
230 NEXT I                              ┘
240 A=PEEK(&HE04F) ←— 有無（００かＦＦ）のデータを読む
250 LOCATE 3,10
260 IF A=0 THEN PRINT "ﾆ ｱﾘﾏｽ"
270 IF A=255 THEN PRINT"ﾆ ｱﾘﾏｾﾝ"
280 LOCATE 0,20
290 END
```

ックの扱いになっているのでＣＬＳ３は必要です。第２行目から第４行目は調べようとする漢字のコードを読み出して来るものです。第５行目で画面に漢字を書いています。600×200のモードなので漢字は大きく表示されています。第６行目のＬＯＣＡＴＥ６，３は次の第７行目のＰＲＩＮＴ文の打ち始めの位置を指定するものです。第８行目のMは漢字データの個数を読み出しています。第９行目から第14行目のループは次々に漢字データのリストを打ち出すためのものです。第15行目は機械語プログラムの実行結果をもらってくるものです。第16行目から第18行目は結果を画面にわかりやすく表示するための工夫です。第19行目はＯｋとカーソルがうるさくなるので、つけておきました。

以下にＢＡＳＩＣプログラムの実行結果を示してあります。

このプログラムもＢＡＳＩＣと機械語のプログラムを別々に作るのでなく、ＢＡＳＩＣのプログラムの中から機械語のプログラムを呼び出して実行させ、再びＢＡＳＩＣプログラムに戻るという形にできたら便利だとは思いませんか。もう少し先でそれを勉強することにしましょう。

```
葦 トイウ カンジ ハ ツギノ リスト

姶逢葵茜穐悪握渥葦芦

ニ アリマス
```

プログラム実行後のメモリの様子

```
E030 31 30 0A 28 30 29 30 2A 30 2B 30 2C 30 2D 30 2E
E040 30 2F 30 31 30 32 30 00 00 00 00 00 00 00 00 00
```

見つかったので００が入る

7・3　次々とデータをおきかえること

擬似命令

ＰＣ−８８０１のアセンブラは大変便利なのですが、擬似命令(Pseudo Operation)が使えないのが不便なところです。擬似命令とはＤＡＴＡやＥＱＵＡＴＥやＤＥＦＩＮＥやＯＲＩＧＩＮやＲＥＳＥＲＶＥなどが代表的なものです。もう１つはラベルを使うことができないことが残念です。N88−ＢＡＳＩＣではラベルが使えるのにアセンブラではどうして使えないのでしょう。全く残念です。ＥＱＵＡＴＥ文が使えれば、よくでて来るアドレスはラベルをつけておいてやれば、アセンブラが自動的に翻訳してくれるので楽なのです。アセンブラがどういう動作をするかは本書の取扱いの範囲を越えますが、アセンブラはアセンブリ言語プログラムを機械語プログラムに変換する際に、記号(symbol)の表を作ります。この表にはアセンブリ言語のプログラムの中で使われている記号がすべてのせられています。この表を参照しながら、２回目のパスのときにはアセンブラはアセンブル作業を実行していくのです。

このプログラムでは＄Ｅ０２０，Ｅ０２１番地に、あるデータをしまっておき、＄Ｅ０２２，＄Ｅ０２３番地にリスト中の第１番目の要素のアドレスをしまい、次々に、次の要素のアドレスをたぐっていき、そのアドレスの中味を＄Ｅ０２０，Ｅ０２１番地の中味におきかえていくというやり方をとっています。

アドレス	内容
Ｅ０００	メインプログラム
Ｅ０１３	未使用
Ｅ０２０ Ｅ０２１	データ
Ｅ０２２ Ｅ０２３	リスト中の第１番目の要素のアドレス
	∫
	リスト中の第ｎ番目の要素のアドレス
	∫
	００　終了記号

プログラムは大変簡単です。まず第１行目でＨＬレジスタに＄Ｅ０２０番地，＄Ｅ０２１番地の中味をしまっています。第２行目，第３行目ではＢレジスタにＨＬレジスタの中味を，ＣレジスタにＬレジス

```
E000    2A E020     LHLD E020  ← データをＨＬレジスタへ
E003    44          MOV  B,H   ← ＄Ｅ０２１番地の中味をＢレジスタへ
E004    4D          MOV  C,L   ← ＄Ｅ０２０番地の中味をＣレジスタへ
E005    2A E022     LHLD E022  ← リスト中の要素の番地をＨＬレジスタへ
E008    5E          MOV  E,M   ← その番地の中味をＥレジスタへ
E009    71          MOV  M,C   ← データを新しい番地へ
E00A    23          INX  H
E00B    56          MOV  D,M
E00C    70          MOV  M,B   ← データを１つ先の番地へ
E00D    EB          XCHG       ← ポインタの切り換え
E00E    7C          MOV  A,H  ┐
E00F    B5          ORA  L    ┘← 処理終了かどうかの判断
E010    C2 E008     JNZ  E008
E013    76          HLT
```

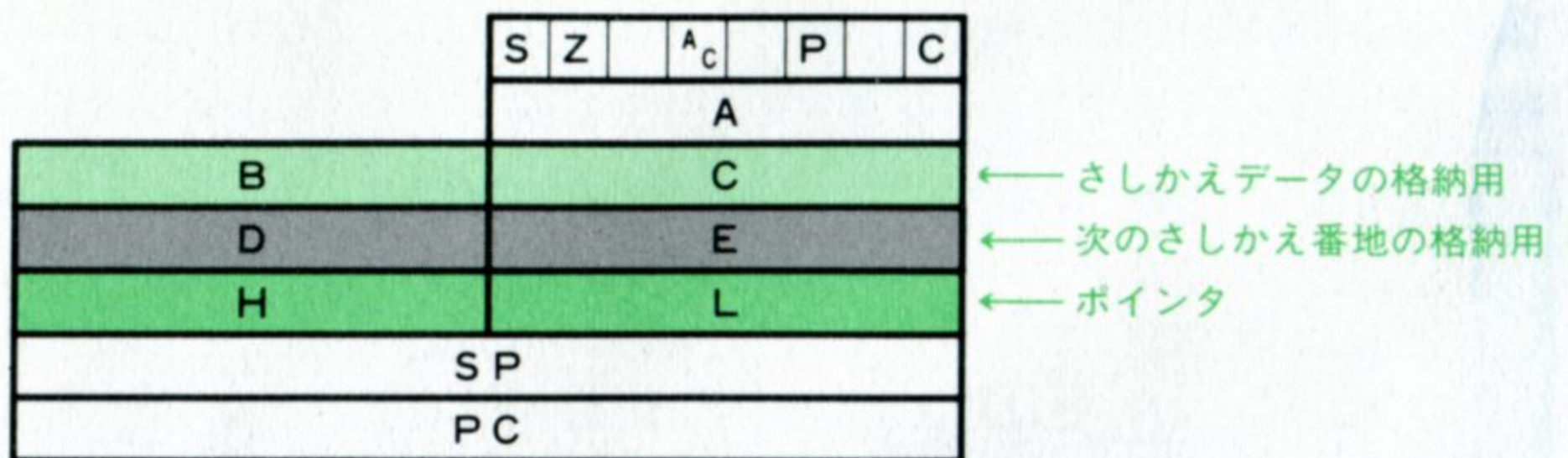
タの中味をうつします。これでB、Cレジスタに＄Ｅ０２０，Ｅ０２１番地のデータが入ったわけです。第４行目はポインタを＄Ｅ０２２番地に指定し、第５行目でポインタの中味をＥレジスタに入れ、第６行目でＣレジスタの中味をポインタの指定する番地にしまっています。つまりＣレジスタの中味が新しい番地へうつりました。

第７行目でポインタの値を１だけすすめ、第８，９行目でポインタの指し示す番地（最初は＄Ｅ０２３番地です）の中味をＤレジスタに代入し、さらにＢレジスタの中味をポインタの指し示す番地、つまり新しい番地のもう１つ先へうつしました。

第10行目のＸＣＨＧはＤＥレジスタとＨＬレジスタの中味を入れかえるものです。つまりポインタをＨＬレジスタの値から、ＤＥレジスタの値にきりかえており、これで次々にデータを追いかけることが可能になります。

第11，12行目の　ＭＯＶ Ａ，Ｈ と ＯＲＡ Ｌは、ＨＬレジスタの中味とＬレジスタの中味のＯＲ演算をしています。これはＨレジスタとＬレジスタの値が両方とも０であるかどうかを調べるための準備なのです。

第13行目の　ＪＮＺ Ｅ００８はＨレジスタとＬレジスタの中味が両方とも０（終了記号）でない場合はＥ００８からの処理のループをくり返すことになります。もし終了記号にぶつかっていれば第14行目のＨＬＴで終わりです。

このプログラムは、レジスタの使い方が大変ややこしいので図にしておきましょう。

プログラム実行前のメモリの様子

プログラム実行後のメモリの様子

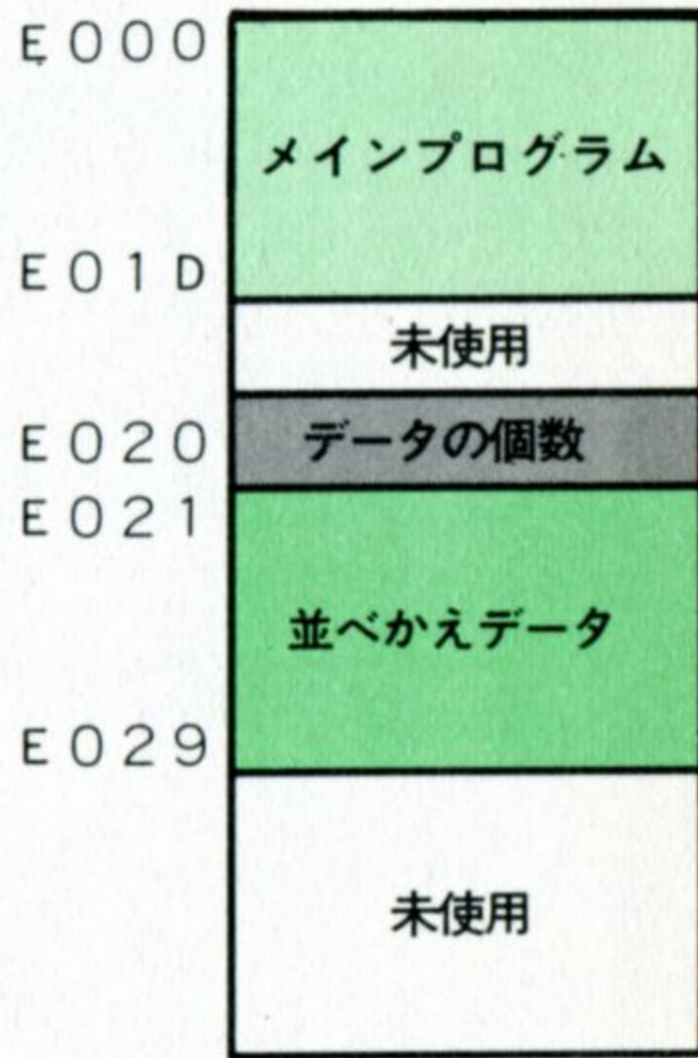
7·4 簡単な並べかえ

ソート
バブル・ソート

ビジネスにおいてもっとも大切なデータ処理の手法にソートとよばれる並べかえの操作があります。たとえば、取引先の名簿を作るために50音順に会社の名前を並べかえるとか、成績高順に会社の名前を並べるとか、納期順に並べかえるとかいろいろな応用があります。また学校等では学生名簿を作ったり、成績順に並べかえたり、なかなかいろいろな応用があるようです。ソートの手法はよく研究されていて、それだけで一冊の本が楽に書けてしまうほどで、いろいろ高級なテクニックもありますが、ここでは一番やさしいバブル・ソートというテクニックだけを勉強しましょう。

E000	メインプログラム
E01D	未使用
E020	データの個数
E021	並べかえデータ
E029	未使用

プログラムを見てください。まず第1行目でBレジスタに0を代入しています。このプログラムではBレジスタを並べかえが行なわれたかどうかの表示用に使います。つまりBレジスタが0なら並べかえがあり、Bレジスタが1なら並べかえがなかったことをあらわします。

第2行目でポインタの値を＄Ｅ０２０番地にしています。第3行目でポインタの指し示す番地つまり＄Ｅ０２０番地の中味の並べかえのデータの個数をCレジスタにしまいます。第4行目でデータの個数を1つへらします。第5行目でポインタの値を1だけすすめ、第6行目

```
E000    06 00       MVI   B,00   ←―― Bレジスタをクリア
E002    21 E020     LXI   H,E020
E005    4E          MOV   C,M    ←―― データ数のセット
E006    0D          DCR   C      ←―― データ数を1へらす
E007    23          INX   H      ←―― ポインタを1すすめる
E008    7E          MOV   A,M    ←―― データをアキュムレータへ
E009    23          INX   H      ←―― ポインタを次のデータのところへ
E00A    BE          CMP   M      ←―― 2数を比較
E00B    D2 E015     JNC   E015
E00E    56          MOV   D,M   ┐
E00F    77          MOV   M,A   ├―― 2数を並べかえる
E010    2B          DCX   H     │
E011    72          MOV   M,D   ┘
E012    23          INX   H
E013    06 01       MVI   B,01   ←―― ソートありの判定
E015    0D          DCR   C
E016    C2 E008     JNZ   E008
E019    05          DCR   B
E01A    CA E000     JZ    E000
E01D    76          HLT
```

でアキュムレータAにポインタの指し示す番地のデータを代入します。第7行目でポインタを1だけすすめ、第8行目のCMP命令で比較をします。

たびたび出てきましたように、CMP命令ではアキュムレータAの中味からポインタの指し示す番地の中味を引いて、その結果、アキュムレータAの中味をこわすことなく、フラグだけを変更するのでした。アキュムレータAの中味がポインタの指し示す番地の中味より小さくない（大きいか等しい）とき、キャリーフラグCの値は0です。アキュムレータAの中味がポインタの指し示す番地の中味より小さいときにはキャリーフラグは1になるのです。そこで、キャリーフラグCは次のように変化します。

先行するデータ≧新しいデータ　　C＝0

先行するデータ＜新しいデータ　　C＝1

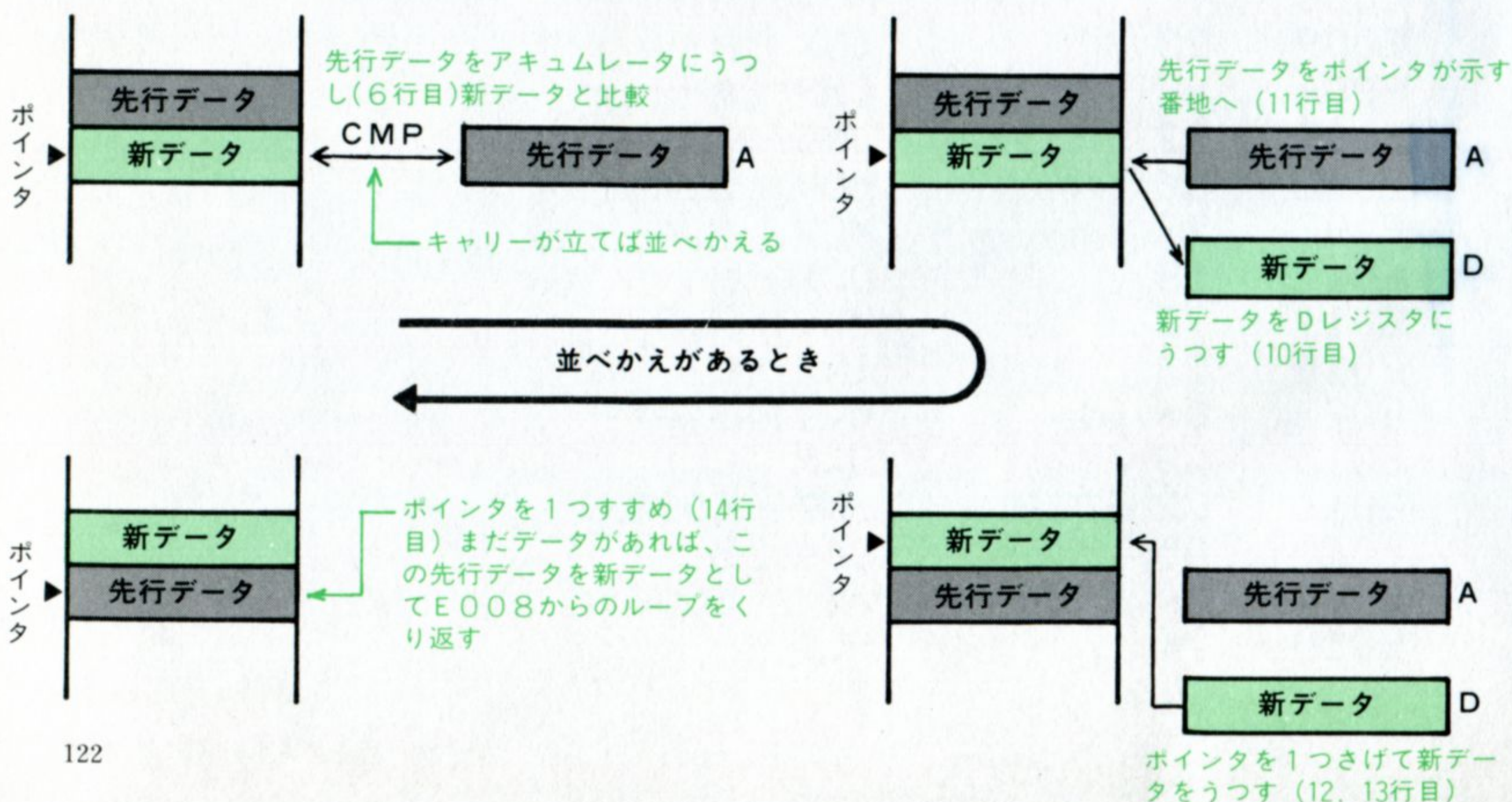
第9行目の　JNC E015　は、キャリーフラグCが0であれば$E015番地へ飛びなさいということで、先行データが新しいデータより小さくないときは$E015番地へ飛ぶことになります。もしキャリーフラグCが1であれば、先行データより新しいデータのほうが大きいのですから、第10行目からの操作で並べかえをすることになります。まず第10行目で新しいデータをDレジスタに入れ、第11行目でアキュムレータAに残っている先行するデータをポインタの指し示す番地にしまいます。第12行目でポインタの値を1だけもどし、第13行目でDレジスタの中味をポインタの更新された値の指し示す番地にしまい、さらに第14行目でポインタの値を1だけすすめています。図解すると下図のようになります。

先行データ
ポインタ▶ 新データ
CMP
先行データ A
先行データをアキュムレータにうつし（6行目）新データと比較
キャリーが立てば並べかえる

先行データ
ポインタ▶ 新データ
先行データ A
新データ D
先行データをポインタが示す番地へ（11行目）
新データをDレジスタにうつす（10行目）

並べかえがあるとき

ポインタ▶ 新データ
先行データ
先行データ A
新データ D
ポインタを1つさげて新データをうつす（12, 13行目）

新データ
ポインタ▶ 先行データ
ポインタを1つすすめ（14行目）まだデータがあれば、この先行データを新データとしてE008からのループをくり返す

そして、この場合並べかえてありましたので、第15行目の　ＭＶＩ　Ｂ，０１　でレジスタBの中味を１とし、並べかえのあったことを表示します。第16行目でデータの処理が１つ終了しているのでCレジスタの値を１だけ減少させます。第17行目の　ＪＮＺ　Ｅ００８　は最後のデータまで処理が終了しているかどうかを調べます。終了していなければＥ００８からのループをくり返します。終了している場合には第18行目の　ＤＣＲ　Ｂ　にすすみます。もし並べかえがあった場合はBレジスタは１ですから、ＤＣＲ　ＢでBレジスタの値は0となり、第19行目のＪＺ　Ｅ０００で再度Ｅ０００からの処理を行なって行きます。並べかえがないときにはBレジスタの値は0ですから、ＤＣＲ　Ｂ　をするとBレジスタの値は0でなく、もはや第19行目の　ＪＺ　Ｅ０００　にはひっかからず終了になります。

プログラム実行前のメモリの様子

```
E020 0A 01 02 03 04 05 06 07 08 09 0A
```
データ数（0A）　データ（01〜0A）

プログラム実行後のメモリの様子

```
E020 0A 0A 09 08 07 06 05 04 03 02 01
```
大きい順に並べかえられている

並べかえのプログラムをＢＡＳＩＣで作ってみましょう。行番号１００から１９０までが並べかえの実際にたずさわっており、行番号２１０から２８０の間が結果の表示を担当する部分です。いかにも簡単そうなプログラムで結果も正しく表示されています。これならＢＡＳＩＣのプログラムだけでアセンブリ言語でプログラムを組む必要がなさそうにみえます。けれども本当にそうでしょうか？

```
100 DIM D(100)
110 READ N
120 FOR I=1 TO N:READ D(I):NEXT I
130 SW=0
140 FOR I=1 TO N-1
150 IF D(I)>=D(I+1) THEN GOTO 180
160 SWAP D(I),D(I+1)
```

```
170 SW=1
180 NEXT I
190 IF SW=1 THEN GOTO 130
200 '
210 CLS 3
220 RESTORE
230 READ N:PRINT "DATA NUMBER IS = ";N
240 PRINT "ORIGINAL DATA "
250 FOR I=1 TO N:READ D:PRINT D;:NEXT I
260 PRINT:PRINT
270 PRINT "SORTED DATA "
280 FOR I=1 TO N:PRINT D(I);:NEXT I
290 '
300 END
310 '
320 DATA 10,1,2,3,4,5,6,7,8,9,10
```

このプログラムを実行すると次のようになります。

```
DATA NUMBER IS =  10
ORIGINAL DATA
 1  2  3  4  5  6  7  8  9  10

SORTED DATA
 10  9  8  7  6  5  4  3  2  1
```

ＢＡＳＩＣプログラムを組むのに時間がかからないとしても、データの処理速度はどうでしょうか。それを調べるプログラムを作ってみましょう。複雑にならないようにＢＡＳＩＣだけで作ってありますので、完全に厳密なものではありませんが、1つの目安になるでしょう。

アセンディング・ソート
ディセンディング・ソート

ＤＡＴＡ文に62個のデータを並べてあります。これまでお話してありませんが、並べかえにはアセンディング・ソートとディセンディング・ソートというものがあります。アセンディング・ソートは小さい順に並べかえ、ディセンディング・ソートは大きい順に並べかえるものです。現在のプログラムはディセンディング・ソートを行なうものですから、データ文中のデータが小さい順に並んでいるので一番時間がかかるようになっています。

62個のデータを並べかえるＢＡＳＩＣプログラムを走らせてみますと54秒かかることがわかります。相当イライラさせられます。

```
100 TIME$="00:00:00"
110 DIM D(100)
120 READ N
130 FOR I=1 TO N:READ D(I):NEXT I
140 SW=0
150 FOR I=1 TO N-1
160 IF D(I)>=D(I+1) THEN GOTO 190
170 SWAP D(I),D(I+1)
180 SW=1
190 NEXT I
200 IF SW=1 THEN GOTO 140
210 '
220 LPRINT "TIME IS = ";RIGHT$(TIME$,2);" SECONDS"
230 LPRINT
240 RESTORE
250 READ N:LPRINT "DATA NUMBER IS = ";N
260 LPRINT "ORIGINAL DATA "
270 FOR I=1 TO N:READ D:LPRINT D;:NEXT I
280 '
290 LPRINT:LPRINT
300 LPRINT "SORTED DATA "
310 FOR I=1 TO N:LPRINT D(I);:NEXT I
320 '
330 END
340 '
350 DATA 62,1,2,3,4,5,6,7,8,9,10,11,12,1
    3,14,15
360 DATA 16,17,18,19,20,21,22,23,24,25,26
    ,27,28,29,30,31
370 DATA 32,33,34,35,36,37,38,39,40,41,42
    ,43,44,45,46,47
380 DATA 48,49,50,51,52,53,54,55,56,57,58
    ,59,60,61,62
```

```
TIME IS = 01 SECONDS
DATA NUMBER IS =  10
ORIGINAL DATA
 1  2  2  3  5  6  7  8  9  10

SORTED DATA
 10  9  8  7  6  5  3  2  2  1
```

データが62個になると54秒もかかります。

```
TIME IS = 54 SECONDS

DATA NUMBER IS =  62
ORIGINAL DATA
  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22
 23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42
 43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62

SORTED DATA
 62  61  60  59  58  57  56  55  54  53  52  51  50  49  48  47  46  45  44  43
 42  41  40  39  38  37  36  35  34  33  32  31  30  29  28  27  26  25  24  23
 22  21  20  19  18  17  16  15  14  13  12  11  10   9   8   7   6   5   4   3   2   1
```

次に機械語プログラムとＢＡＳＩＣプログラムをつないで、実行速度を比較してみましょう。このプログラムはＢＡＳＩＣプログラムの中から機械語プログラムを呼んでいるものです。ＰＣ-８８０１では、ＢＡＳＩＣプログラムの中から機械語プログラムを呼び出す方法がいろいろ準備されています。その中で最も使いやすいＣＡＬＬ文の技巧を使ってみましょう。ＢＡＳＩＣプログラムの中で機械語プログラムの開始番地を宣言し、必要なときにＣＡＬＬ文で呼べばよいのです。実際の使い方は次のプログラムを見てください。

ＣＡＬＬ文

ＢＡＳＩＣプログラム

```
100 SORT=&HE000
110 TIME$="00:00:00"
120 CALL SORT
130 LPRINT "TIME IS = ";RIGHT$(TIME$,2);"
    SECONDS"
140 '
150 LPRINT:LPRINT
160 N=PEEK(&HE020)
170 FOR I=1 TO N:LPRINT HEX$(PEEK(&HE020));NEXT I
180 '
190 END
```

機械語部分のプログラムは121ページと同じですが、最終行だけＲＥＴＵＲＮ命令にかえます。

リストを次ページに示します。

```
E000    06 00       MVI  B,00
E002    21 E020     LXI  H,E020
E005    4E          MOV  C,M
E006    0D          DCR  C
E007    23          INX  H
E008    7E          MOV  A,M
E009    23          INX  H
E00A    BE          CMP  M
E00B    D2 E015     JNC  E015
E00E    56          MOV  D,M
E00F    77          MOV  M,A
E010    2B          DCX  H
E011    72          MOV  M,D
E012    23          INX  H
E013    06 01       MVI  B,01
E015    0D          DCR  C
E016    C2 E008     JNZ  E008
E019    05          DCR  B
E01A    CA E000     JZ   E000
E01D    C9          RET  ←――――― ここだけＨＬＴから変わります
```

はじめにデータが62個の場合、次にデータが255個の場合を実験してみました。62個の場合は１秒とかかっていません。255個の場合は１秒です。大変高速であることがわかります。これだけ明らかなスピードの違いがありますと機械語プログラムが少々面倒でも採用しないわけにはいかないことがおわかりと思います。

データ数62のときのプログラム実行前のメモリの様子

```
E020 3E 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
E030 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
E040 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
E050 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 00
```

実行結果

```
TIME IS = 00 SECONDS

62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43
42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23
22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

データ数255のときのプログラム実行前のメモリの様子

```
E020 FF 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
E030 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
E040 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
E050 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
E060 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
E070 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
E080 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F
E090 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
E0A0 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
E0B0 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
E0C0 A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF
E0D0 B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
E0E0 C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF
E0F0 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
E100 E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF
E110 F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
```

実行結果

```
TIME IS = 01 SECONDS

 255  254  253  252  251  250  249  248  247  246  245  244  243  242  241  240
 239  238  237  236  235  234  233  232  231  230  229  228  227  226  225  224
 223  222  221  220  219  218  217  216  215  214  213  212  211  210  209  208
 207  206  205  204  203  202  201  200  199  198  197  196  195  194  193  192
 191  190  189  188  187  186  185  184  183  182  181  180  179  178  177  176
 175  174  173  172  171  170  169  168  167  166  165  164  163  162  161  160
 159  158  157  156  155  154  153  152  151  150  149  148  147  146  145  144
 143  142  141  140  139  138  137  136  135  134  133  132  131  130  129  128
 127  126  125  124  123  122  121  120  119  118  117  116  115  114  113  112
 111  110  109  108  107  106  105  104  103  102  101  100  99   98   97   96   95
 94   93   92   91   90   89   88   87   86   85   84   83   82   81   80   79   78   77   76   75
 74   73   72   71   70   69   68   67   66   65   64   63   62   61   60   59   58   57   56   55
 54   53   52   51   50   49   48   47   46   45   44   43   42   41   40   39   38   37   36   35
 34   33   32   31   30   29   28   27   26   25   24   23   22   21   20   19   18   17   16   15
 14   13   12   11   10   9   8   7   6   5   4   3   2   1
```

もっとデータの個数が多い場合にはどうしたら良いでしょう。いろいろ考えてみてください。データの個数が多くなってきた場合には、クイック・ソートとかいろいろ高級な技巧を使います。

7・5 キーワードとジャンプテーブルを使う

ジャンプテーブル

キーワード

ジャンプテーブルというのは、ある与えられたキーワードと照合させて、そのキーワードに対応するジャンプ先を決めるためのものです。ＢＡＳＩＣそのものでも、これをうまく利用しています。たとえば、ＩＮＰＵＴというＢＡＳＩＣのキーワードが与えられると、これはジャンプテーブルによって対応する機械語処理ルーチンへとジャンプすることになります。

簡単な例題を勉強しましょう。第１行目でＢレジスタにジャンプテーブルのデータの個数26を入れています。もちろん英文字のＡからＺに対応するものであることがわかります。第２行目でアキュムレータＡに＄Ｅ０２０番地へのエントリー・キーワードを代入します。第３行目でポインタを＄Ｅ０２１番地に指定します。第４行目でエントリー・キーワードとジャンプテーブルのデータを比較します。第５行目でポインタを１だけすすめてから結果の判定をしています。

ジャンプテーブルのデータは３バイトでできており、最初の１バイトがキーワード、次の２バイトがジャンプ先のアドレスということになっているので、ポインタを１すすめておいたほうがプログラムが作りやすくなるのです。

第７，８行目でポインタを２だけすすめ、第９行目でＢレジスタの値を１だけ減らします。データがつきれば第10行目のＪＺで終了とな

アドレス	内容
E000	メインプログラム
E01A	未使用
E020	キーワード
E021	●キーワード ●ジャンプテーブル
E06E	未使用

```
E000    06 1A      MVI   B,1A    ←── データ数をBレジスタへ
E002    3A E020    LDA   E020    ←── キーワードをアキュムレータへ
E005    21 E021    LXI   H,E021  ←── ポインタをセット
E008    BE         CMP   M       ←── エントリー・キーワードとデータの比較
E009    23         INX   H
E00A    CA E016    JZ    E016    ←── 見つかればジャンプ
E00D    23         INX   H
E00E    23         INX   H
E00F    05         DCR   B       ←── カウンタを１へらす
E010    CA E01A    JZ    E01A    ←── データがなくなれば終わり
E013    C3 E008    JMP   E008
E016    5E         MOV   E,M  ┐
E017    23         INX   H    ├── ジャンプ先のアドレスをDEレジスタへ
E018    56         MOV   D,M  ┘
E019    EB         XCHG          ←── DE，HLレジスタの中味を入れかえる
E01A    76         HLT
```

ります。データが残っていれば、第11行目のＪＭＰにすすみます。
　＄Ｅ００８番地から＄Ｅ０１５番地のループの抜け方には２通りありますが、第６行目のＪＺで、＄Ｅ０１６番地へとび出すと、ポインタのさしている番地の中味をＤ、Ｅレジスタへうつします。つまりＤＥレジスタにはジャンプ先がしまわれるのです。第15行目のＸＣＨＧでＤＥレジスタの中味とＨＬレジスタの中味が入れかえられます。第16行目にジャンプコマンドを入れておくとジャンプするのですが、暴走の危険がありますので、ここでやめておきます。結果はＸコマンドでＨＬレジスタの中味を見ることで確認することにしました。


```
E020 45 41 8A 6B 42 A0 6B 43 AF 6B 44 0E 6C 45 4A 6C
E030 46 6F 6C 47 89 6C 48 9B 6C 49 A4 6C 4A CE 6C 4B
E040 CF 6C 4C DC 6C 4D 1C 6D 4E 40 6D 4F 4F 6D 50 68
E050 6D 51 97 6D 52 98 6D 53 DA 6D 54 21 6D 55 41 6E
E060 56 4A 6E 57 58 6E 58 7B 6E 59 7F 6E 5A 80 6E 00
```

●プログラム実行後のレジスタの様子


```
A :45 F :PZ---ON- B :1600 D :E02F H :6C4A
H´:0000 IX:1008 IY:0000 I :F3 PC:E01B SP:DFFB
A´:00 F´:P----O-C B´:8000 D´:0080
```

7・6 PC-8801のBASICを探検する

世の中にはＰＣ－８００１やＰＣ－８８０１のＢＡＳＩＣのＲＯＭの中味を克明に解析した本がたくさんでています。私も可能な限り入手して研究しますが、大変な努力だと感心します。ＢＡＳＩＣの中味を研究することは我々にはできないでしょうか。我々も１つＰＣ－8801のＢＡＳＩＣの中味を探検してみましょう。もちろんすべてを解明するなどと大胆なことは考えないことにします。必要なのは探検用のルートを作ることだけです。それに解析書も何冊か手に入りますので、ここではルートだけ作って我慢することにします。

ＰＣ－８８０１のＢＡＳＩＣはマイクロソフト社によって作られたといわれています。私もマイクロソフト社のＢＡＳＩＣにはいつも感心させられ、研究してきています。なつかしいコモドールのＣＢＭ３０３２はいまも私の手もとで現役ですし、ラジオシャックのＴＲＳ８０もときどき動いています。ＰＣ－８００１や日立のレベルⅢや沖のｉｆ８００、富士通のＦＭ８など、だんだん複雑になっていく過程が明らかです。ＰＣ－８８０１は一層の改良の跡が見られ調べにくい部分もありますがＰＣ－８８０１の強力なモニタプログラムの力を借りると徐々に中味がわかってきます。

同じ会社の作ったソフトウェアは必ずくせのあるものです。そこでマイクロソフト社のＢＡＳＩＣのくせを理解すると、大体は解読できるものです。調べ方のコツは、モニタプログラムのＤコマンドを使って拡大なＢＡＳＩＣのメモリ領域を調べていくだけです。まずＤ００００として一行表示させ、⏎キーを押しつづけると、次々にメモリの内容が表示されていきます。これをメモリ領域のすべてにわたって行ないます。大変そうに思えますが、そんなことはありません。数分間で全部見ることができます。これを２、３回くり返して中味の様子を頭に入れます。おそらく気がつかれることと思いますが、ある所に来ると急に右端のＡＳＣＩＩコードの表示部分に読みやすい英語がでてくることがあります。これは何個所かあるはずです。この中で１個所だけが特に大切です。

マイクロソフト社のＢＡＳＩＣの作り方を見ていますと、ふつうある部分でＢＡＳＩＣのキーワード(ＥＮＤ，ＦＯＲ，ＮＥＸＴなど)に対する中間言語に対するポインタを整然と並べ、次にＢＡＳＩＣのキーワードを並べる方式と、ＢＡＳＩＣのキーワードに対応する中間言語のポインタを整然と並べ、これに対応するジャンプテーブルという解読用の表を配置し、さらにキーワードと中間言語の対応表を並べる

キーワード
中間言語
ジャンプテーブル

方式があるようです。いずれにしてもどこかにキーワードがずらりと並んでいる所があるはずです。これさえ探しあてれば、その前後にＢＡＳＩＣのキーワードに対する中間言語というもののポインタの表があり、ジャンプテーブルがあるはずです。糸口はここにあります。そこで簡単なＢＡＳＩＣプログラムを使って、有望と思われる部分から調べ始めます。

モニタプログラムですぐわかることは＆Ｈ６Ｂ８Ａ番地からＢＡＳＩＣのキーワードらしきものが並んでいることです。何だかよくわからないものもありますが、これだけわかれば十分です。あとはＰＣ-８８０１を最大限利用して調べていけばよいのです。

```
6B80 4A 6E 58 6E 7B 6E 7F 6E 80 6E 55 54 CF A8 4E C4   JnXn{n n▁nUTﾏｨNﾄ
6B90 F8 42 D3 06 54 CE 0E 53 C3 15 54 54 52 A4 EB 00    Bﾓ Tﾎ Sﾃ TTR､♣
6BA0 53 41 56 C5 D5 4C 4F 41 C4 D4 45 45 D0 D7 00 4F   SAVﾅｺLOAﾄﾔEEﾐﾗ O
6BB0 4E 53 4F 4C C5 9D 4F 50 D9 CD 4C 4F 53 C5 C0 4F   NSOLﾅ､OPﾙﾍLOSﾅﾀO
6BC0 4E D4 99 4C 45 41 D2 92 53 52 4C 49 CE 54 49 4E   Nﾔ┐LEAﾒ┤SRLIﾎTIN
6BD0 D4 1C 53 4E C7 1D 44 42 CC 1E 56 C9 20 56 D3 21   ﾔ SNﾇ DB7 Vﾉ Vﾓ!
6BE0 56 C4 22 4F D3 0C 48 52 A4 16 41 4C CC B1 4F 4D   Vﾄﾟ Oﾓ HR､ ALﾌｱOM
6BF0 4D 4F CE B6 48 41 49 CE B7 4F CD 5A 49 52 43 4C   MOﾎｶHAIﾎｷOﾍZIRCL
6C00 C5 CC 4F 4C 4F D2 CB 4C D3 CE 4D C4 64 00 45 4C   ﾅﾌOLOﾒﾋLﾓﾎMﾄd EL
6C10 45 54 C5 A7 41 54 C1 84 49 CD 86 45 46 53 54 D2   ETﾅｧATﾁ▄Iﾍ■EFSTﾒ
6C20 AA 45 46 49 4E D4 AB 45 46 53 4E C7 AC 45 46 44   ｪEFINﾔｫEFSNﾇｬEFD
6C30 42 CC AD 53 4B 4F A4 BA 45 C6 97 53 4B 49 A4 EC   BﾌｭSKO､ｺEﾆ ﾚKI､●
6C40 53 4B C6 50 41 54 45 A4 59 00 4C 53 C5 9F 4E C4   SKﾆPATE､Y LSﾅﾞNﾄ
6C50 81 52 41 53 C5 A3 44 49 D4 A4 52 52 4F D2 A5 52   ▁RASﾅ｣DIﾔ､RROﾒ･R
6C60 CC E4 52 D2 E5 58 D0 0B 4F C6 23 51 D6 FB 00 4F   ﾌ◢Rﾒ◣Xﾐ Oﾆ#Qﾖ  O
6C70 D2 82 49 45 4C C4 BC 49 4C 45 D3 C3 CE E1 52 C5   ﾒ▄IELﾄｼILEﾓﾃﾎ ﾊRﾅ
6C80 0F 49 D8 1F 50 4F D3 26 00 4F 54 CF 89 4F 20 54    Iﾘ POﾓ& OTﾏ▌O T
6C90 CF 89 4F 53 55 C2 8D 45 D4 BD 00 45 58 A4 1A 45   ﾏ▌OSUﾂ■Eﾔｽ EX､ E
6CA0 4C D0 D9 00 4E 50 55 D4 85 53 45 D4 60 45 45 C5   Lﾐﾙ NPUﾔ▄SEﾔ`EEﾅ
6CB0 61 52 45 53 45 D4 62 C6 8B 4E 53 54 D2 E8 4E D4   aRESEﾔbﾆ▌NSTﾒ♠Nﾔ
6CC0 05 4E D0 10 4D D0 FC 4E 4B 45 59 A4 EF 00 00 45    Nﾐ Mﾐ NKEY､＼ E
6CD0 D9 5B 49 4C CC C5 41 4E 4A C9 DB 00 4F 43 41 54   ﾙ[ILﾌﾅANJﾉﾛ OCAT
6CE0 C5 D6 50 52 49 4E D4 9B 4C 49 53 D4 9C 50 4F D3   ﾅﾖPRINﾔ┘LISﾔ┌POﾓ
6CF0 1B 45 D4 88 49 4E C5 AE 4F 41 C4 C1 53 45 D4 C6    Eﾔ▌INﾅｮOAﾄﾁSEﾔﾆ
6D00 49 53 D4 93 46 49 4C 45 D3 C9 4F C7 0A 4F C3 24   ISﾔ├FILEﾓﾉOﾇ Oﾃ$
6D10 45 CE 12 45 46 54 A4 01 4F C6 25 00 4F 54 4F D2   Eﾎ EFT､ Oﾆ% OTOﾒ
6D20 57 45 52 47 C5 C2 4F C4 FD 4B 49 A4 27 4B 53 A4   WERGﾅﾂOﾄ KI､'KS､
6D30 28 4B 44 A4 29 49 44 A4 03 4F CE CA 41 D0 55 00   (KD､)ID､ OﾎﾊAﾐU
6D40 45 58 D4 83 41 4D C5 C4 45 D7 94 4F D4 E3 00 50   EXﾔ▄AMﾅﾄEﾗ▔Oﾔ◥ P
6D50 45 CE BB 55 D4 9A CE 95 D2 F9 43 54 A4 19 50 54   EﾎｻUﾔ└ﾎ─ﾒ CT､ PT
6D60 49 4F CE B8 46 C6 EE 00 52 49 4E D4 91 55 D4 BE   IOﾎｸFﾆ╱ RINﾔ┬Uﾔｾ
6D70 4F 4B C5 98 4F 4C CC 5F 4F D3 11 45 45 CB 17 53   OKﾅ┌OLﾌ_Oﾓ EEﾋ S
6D80 45 D4 CF 52 45 53 45 D4 D0 4F 49 4E D4 53 41 49   EﾔﾏRESEﾔﾐOINﾔSAI
6D90 4E D4 D1 45 CE 58 00 00 45 54 55 52 CE 8E 45 41   NﾔﾑEﾎX  ETURﾎ■EA
6DA0 C4 87 55 CE 8A 45 53 54 4F 52 C5 8C 42 59 54 C5   ﾄ■Uﾎ▌ESTORﾅ▌BYTﾅ
6DB0 5E 45 CD 8F 45 53 55 4D C5 A6 53 45 D4 C7 49 47   ^Eﾍ┼ESUMﾅｦSEﾔﾇIG
6DC0 48 54 A4 02 4E C4 08 45 4E 55 CD A9 41 4E 44 4F   HT､ Nﾄ ENUﾍｩANDO
6DD0 4D 49 5A C5 B9 4F 4C CC D8 00 43 52 45 45 CE D3   MIZﾅｹOLﾌﾘ CREEﾎﾓ
6DE0 45 41 52 43 C8 56 54 4F D0 90 57 41 D0 A2 45 D4   EARCﾈVTOﾐ┴WAﾐ｢Eﾔ
6DF0 BF 52 D1 ED 54 41 54 55 D3 63 41 56 C5 C8 50 43   ｿRﾑ◥TATUﾓcAVﾅﾈPC
6E00 A8 E2 54 45 D0 DF 47 CE 04 51 D2 07 49 CE 09 54   ｨ◤TEﾐﾟGﾎ Qﾒ Iﾎ T
6E10 52 A4 13 54 52 49 4E 47 A4 E6 50 41 43 45 A4 18   R､ TRING､◤PACE､
6E20 00 48 45 CE DD 52 4F CE A0 52 4F 46 C6 A1 41 42    HEﾎﾝROﾎ ROFﾆ｡AB
6E30 A8 DE CF DC 41 CE 0D 45 52 CD D2 49 4D 45 A4 5C   ｨﾞﾏﾜAﾎ ERﾍﾒIME､¥
6E40 00 53 49 4E C7 E7 53 D2 E0 00 41 CC 14 49 45 D7    SINﾇ◤Sﾒ= Aﾌ IEﾗ
6E50 51 41 52 50 54 D2 EA 00 49 44 54 C8 9E 49 4E 44   QARPTﾒ◆ IDTﾈ╰IND
```

右端の表示が読みにくいのは、ＡＳＣＩＩの８ビットコードで表示されているので、カタカナが混じるためです。そこで、ＡＳＣＩＩ７ビットコードで読み出せばよいことがわかります。そして、この前後に重要なデータがかたまっていますので、プリンタで打ち出して調べあげてしまいます。結論的にいいますと重要な部分は＆Ｈ６９ＦＥ部分からありますので、全部打ち出すプログラムを作ってみましょう。

●少し力まかせでぎこちない部分があります。

```
100 I=&H69FE
110 J=129
120 '
130 IF I>&H6F11 THEN END
140 '
150 S2=PEEK(I+1)
160 A$=HEX$(S2)
170 IF LEN(A$)<2 THEN A$="0"+A$
180 IF S2<32 THEN C$=" ":GOTO 220
190 IF (S2 AND 127)<32 THEN C$=" ":GOTO 220
200 C$=CHR$(S2 AND 127)
210 '
220 S1=PEEK(I)
230 B$=HEX$(S1)
240 IF LEN(B$)<2 THEN B$="0"+B$
250 IF S1<32 THEN D$=" ":GOTO 290
260 IF (S1 AND 127)<32 THEN D$=" ":GOTO 290
270 D$=CHR$(S1 AND 127)
280 '
290 LPRINT HEX$(I),J,HEX$(J),A$+B$,D$+C$
300 I=I+2:J=J+1
310 GOTO 130
```

●データ１——中間言語と機械語処理ルーチンの対応表を作ること

アドレス	中間言語の10進表示	中間言語の16進表示	機械語ルーチンの所在	ＡＳＣＩＩ表示
69FE	129	81	50E5	eP
6A00	130	82	08BF	?
6A02	131	83	52BD	=R
6A04	132	84	0C77	w
6A06	133	85	102D	−
6A08	134	86	5AC5	EZ
6A0A	135	87	10F9	y
6A0C	136	88	0C9C	
6A0E	137	89	0BF9	y
6A10	138	8A	0B7C	¦
6A12	139	8B	0E05	

アドレス	中間言語の10進表示	中間言語の16進表示	機械語ルーチンの所在候補	ASCII表示
6A14	140	8C	50A5	%P
6A16	141	8D	0BBF	?
6A18	142	8E	0C41	A
6A1A	143	8F	0C79	y
6A1C	144	90	50CA	JP
6A1E	145	91	0E54	T
6A20	146	92	522E	.R
6A22	147	93	18D9	Y
6A24	148	94	77DD	]w
6A26	149	95	0D01	
6A28	150	96	1800	
6A2A	151	97	15D7	W
6A2C	152	98	1B84	
6A2E	153	99	5140	@Q
6A30	154	9A	17FA	z
6A32	155	9B	0E4C	L
6A34	156	9C	18D4	T
6A36	157	9D	7071	qp
6A38	158	9E	181A	
6A3A	159	9F	0C79	y
6A3C	160	A0	5158	XQ
6A3E	161	A1	5159	YQ
6A40	162	A2	515E	^Q
6A42	163	A3	519C	Q
6A44	164	A4	657B	{e
6A46	165	A5	0DCA	J
6A48	166	A6	0D8D	
6A4A	167	A7	1B40	@
6A4C	168	A8	0DD5	U
6A4E	169	A9	6F0E	o
6A50	170	AA	0AC4	D
6A52	171	AB	0AC7	G
6A54	172	AC	0ACA	J
6A56	173	AD	0ACD	M
6A58	174	AE	0FAA	*
6A5A	175	AF	EE80	n
6A5C	176	B0	EE83	n
6A5E	177	B1	EE89	n
6A60	178	B2	0000	
6A62	179	B3	0000	
6A64	180	B4	0000	
6A66	181	B5	EE86	n
6A68	182	B6	EEC5	En
6A6A	183	B7	EE7A	zn
6A6C	184	B8	1C89	
6A6E	185	B9	1CD1	Q
6A70	186	BA	EE98	n
6A72	187	BB	4798	G
6A74	188	BC	4A5C	¥J
6A76	189	BD	7198	q
6A78	190	BE	71A6	&q
6A7A	191	BF	EE8C	n
6A7C	192	C0	4B04	K
6A7E	193	C1	4854	TH

アドレス	中間言語の10進表示	中間言語の16進表示	機械語ルーチンの所在候補	ASCII表示
6A80	194	C2	4855	UH
6A82	195	C3	EE9B	n
6A84	196	C4	EE8F	n
6A86	197	C5	EE92	n
6A88	198	C6	49AB	+I
6A8A	199	C7	49AA	*I
6A8C	200	C8	48A3	#H
6A8E	201	C9	EE9E	n
6A90	202	CA	E826	&h
6A92	203	CB	6EC6	Fn
6A94	204	CC	6ECE	Nn
6A96	205	CD	6EA6	&n
6A98	206	CE	71B5	5q
6A9A	207	CF	6E9A	n
6A9C	208	D0	6E96	n
6A9E	209	D1	6EDA	Zn
6AA0	210	D2	7367	gs
6AA2	211	D3	6EDE	^n
6AA4	212	D4	EEBF	?n
6AA6	213	D5	EEC2	Bn
6AA8	214	D6	714E	Nq
6AAA	215	D7	3EB4	4>
6AAC	216	D8	6ECA	Jn
6AAE	217	D9	72AB	+r
6AB0	218	DA	575A	ZW
6AB2	219	DB	578A	W
6AB4	220	DC	5793	W
6AB6	221	DD	20B3	3 .
6AB8	222	DE	2295	
6ABA	223	DF	20A0	
6ABC	224	E0	2E05	.
6ABE	225	E1	2F1A	/
6AC0	226	E2	2F91	/
6AC2	227	E3	1F10	
6AC4	228	E4	2E6E	n.
6AC6	229	E5	2F8B	/
6AC8	230	E6	302C	,0
6ACA	231	E7	EEC8	Hn
6ACC	232	E8	58E4	dX
6ACE	233	E9	17E5	e
6AD0	234	EA	1586	
6AD2	235	EB	56F8	xV
6AD4	236	EC	54CB	KT
6AD6	237	ED	57B4	4W
6AD8	238	EE	5704	W
6ADA	239	EF	5714	W
6ADC	240	F0	1B7A	z
6ADE	241	F1	5741	AW
6AE0	242	F2	54C1	AT
6AE2	243	F3	54C6	FT
6AE4	244	F4	1581	
6AE6	245	F5	21A0	!
6AE8	246	F6	2214	.
6AEA	247	F7	223E	>'

これより上は正しい表示↑

↓これより下は＄DAからでなく＄FF81からに対応している

アドレス	10進表示	16進表示	機械語ルーチンの所在候補	ASCII表示
6AEC	248	F8	2286	.
6AEE	249	F9	4ABA	:J
6AF0	250	FA	4ABD	=J
6AF2	251	FB	4AC0	@J
6AF4	252	FC	4C51	QL
6AF6	253	FD	4C2F	/L
6AF8	254	FE	4C40	@L
6AFA	255	FF	4C62	bL
6AFC	256	100	4AA1	!J
6AFE	257	101	4AA4	$J
6B00	258	102	4AA7	´J
6B02	259	103	EE77	wn
6B04	260	104	0000	
6B06	261	105	6EE2	bn
6B08	262	106	6EBA	:n
6B0A	263	107	6ED2	Rn
6B0C	264	108	6ED6	Vn
6B0E	265	109	6EC2	Bn
6B10	266	10A	6EB2	2n
6B12	267	10B	3F31	1?
6B14	268	10C	0000	
6B16	269	10D	6E9E	n
6B18	270	10E	0000	
6B1A	271	10F	EE7D	}n
6B1C	272	110	0000	
6B1E	273	111	0393	
6B20	274	112	7F30	0
6B22	275	113	6EF2	rn
6B24	276	114	72C8	Hr
6B26	277	115	6F02	o
6B28	278	116	721C	r
6B2A	279	117	0393	
6B2C	280	118	7D71	q}
6B2E	281	119	0393	
6B30	282	11A	6EFA	zn
6B32	283	11B	6EFE	~n
6B34	284	11C	7279	yr
6B36	285	11D	0000	
6B38	286	11E	EEA1	!n
6B3A	287	11F	0000	
6B3C	288	120	EEA4	$n
6B3E	289	121	0000	
6B40	290	122	EEA7	´n
6B42	291	123	0000	
6B44	292	124	EEAA	*n
6B46	293	125	EEB9	9n
6B48	294	126	0000	
6B4A	295	127	0000	
6B4C	296	128	EEAD	-n
6B4E	297	129	EEB3	3n
6B50	298	12A	EEB0	0n
6B52	299	12B	0000	
6B54	300	12C	EEB6	6n
6B56	301	12D	6B8A	k

ここまで$FFA9に対応↓

↓ここから$FFD0以下に対応

●データ2――ジャンプテーブルを作ること（A～Z）

アドレス	10進表示	16進表示		ASCII表示
6B58	302	12E	6BA0	k
6B5A	303	12F	6BAF	/k
6B5C	304	130	6C0E	l
6B5E	305	131	6C4A	Jl
6B60	306	132	6C6F	ol
6B62	307	133	6C89	l
6B64	308	134	6C9B	l
6B66	309	135	6CA4	$l
6B68	310	136	6CCE	Nl
6B6A	311	137	6CCF	Ol
6B6C	312	138	6CDC	¥l
6B6E	313	139	6D1C	m
6B70	314	13A	6D40	@m
6B72	315	13B	6D4F	Om
6B74	316	13C	6D68	hm
6B76	317	13D	6D97	m
6B78	318	13E	6D98	m
6B7A	319	13F	6DDA	Zm
6B7C	320	140	6E21	!n
6B7E	321	141	6E41	An
6B80	322	142	6E4A	Jn
6B82	323	143	6E58	Xn
6B84	324	144	6E7B	{n
6B86	325	145	6E7F	n
6B88	326	146	6E80	n

●データ3――BASICのキーワードと中間言語の対応を探すこと

アドレス	16進表示	ASCII表示
6B8A	5455	UT
6B8C	A8CF	O(
6B8E	C44E	ND
6B90	42F8	xB
6B92	06D3	S
6B94	CE54	TN
6B96	530E	S
6B98	15C3	C
6B9A	5454	TT
6B9C	A452	R$
6B9E	00EB	k
6BA0	4153	SA
6BA2	C556	VE
6BA4	4CD5	UL
6BA6	414F	OA
6BA8	D4C4	DT
6BAA	4545	EE
6BAC	D7D0	PW
6BAE	4F00	O

アドレス	16進表示	ASCII表示
6BB0	534E	NS
6BB2	4C4F	OL
6BB4	9DC5	E
6BB6	504F	OP
6BB8	CDD9	YM
6BBA	4F4C	LO
6BBC	C553	SE
6BBE	4FC0	@O
6BC0	D44E	NT
6BC2	4C99	L
6BC4	4145	EA
6BC6	92D2	R
6BC8	5253	SR
6BCA	494C	LI
6BCC	54CE	NT
6BCE	4E49	IN
6BD0	1CD4	T
6BD2	4E53	SN
6BD4	1DC7	G
6BD6	4244	DB
6BD8	1ECC	L
6BDA	C956	VI
6BDC	5620	V
6BDE	21D3	S!
6BE0	C456	VD
6BE2	4F22	"O
6BE4	0CD3	S
6BE6	5248	HR
6BE8	16A4	$
6BEA	4C41	AL
6BEC	B1CC	L1
6BEE	4D4F	OM
6BF0	4F4D	MO
6BF2	B6CE	N6
6BF4	4148	HA
6BF6	CE49	IN
6BF8	4FB7	7O
6BFA	5ACD	MZ
6BFC	5249	IR
6BFE	4C43	CL

ものすごく長いプリンタ出力と思われるかも知れませんが、それだけのことはあるのです。最初に出てくる&Ｈ６９ＦＥから&Ｈ６Ｂ５５までの長い出力は、ＢＡＳＩＣのキーワードに対応する中間言語に対応する機械語プログラムルーチンの所在場所を示しています。つまり中間言語とＢＡＳＩＣキーワードの対応表さえあれば、ＥＮＤやＦＯＲやＮＥＸＴに対応する機械語プログラムがどこにあるかを教えてくれる大切な表なのです。

次に出てくるのは&Ｈ６Ｂ５６から&Ｈ６Ｂ８９までの表です。非常に整然とした表で、これがジャンプテーブルであることはすぐわか

ります。しかも26行あるのでＡＢＣ順のジャンプテーブルであることもわかります。

続いて出てくるのは&Ｈ６Ｂ８ＡからのＢＡＳＩＣのキーワードの表です。謎のような部分があります。そのままでは意味が通りません。

これら３つのデータを加工していくことを考えましょう。まず中間言語と機械語ルーチンのアドレスの対応表ですが、これはプログラム１、２によって作り出すことができます。中間言語のわりあては&Ｈ８１から&ＨＦＦ，&ＨＦＦ８１から&ＨＦＦＥＦとわりあてられています。

次にジャンプテーブルの処理ですが、これは簡単なＢＡＳＩＣプログラム３で表を作りあげることができます。結果を142ページに示します。

さらにＢＡＳＩＣのキーワードと中間言語の対応表を作ります。これが少しわかりにくいので説明しておきましょう。たとえばデータ３をみてみましょう。

ＵＴＯ(ＮＤｘＢＳ␣ＴＮ␣ＳＣ␣ＴＴＲ＄ｋ␣ＳＡＶＥＬＯＡＤＴＥＥＰ

これは一体何だろうと首をひねったのですが、試みにＡやＢをつけてみますとキーワードが出現します。

ＡＵＴＯ，ＡＮＤ，ＡＴＮ，ＡＳＣ，ＡＴＴＲ＄，ＢＳＡＶＥ，ＢＬＯＡＤ，ＢＥＥＰ

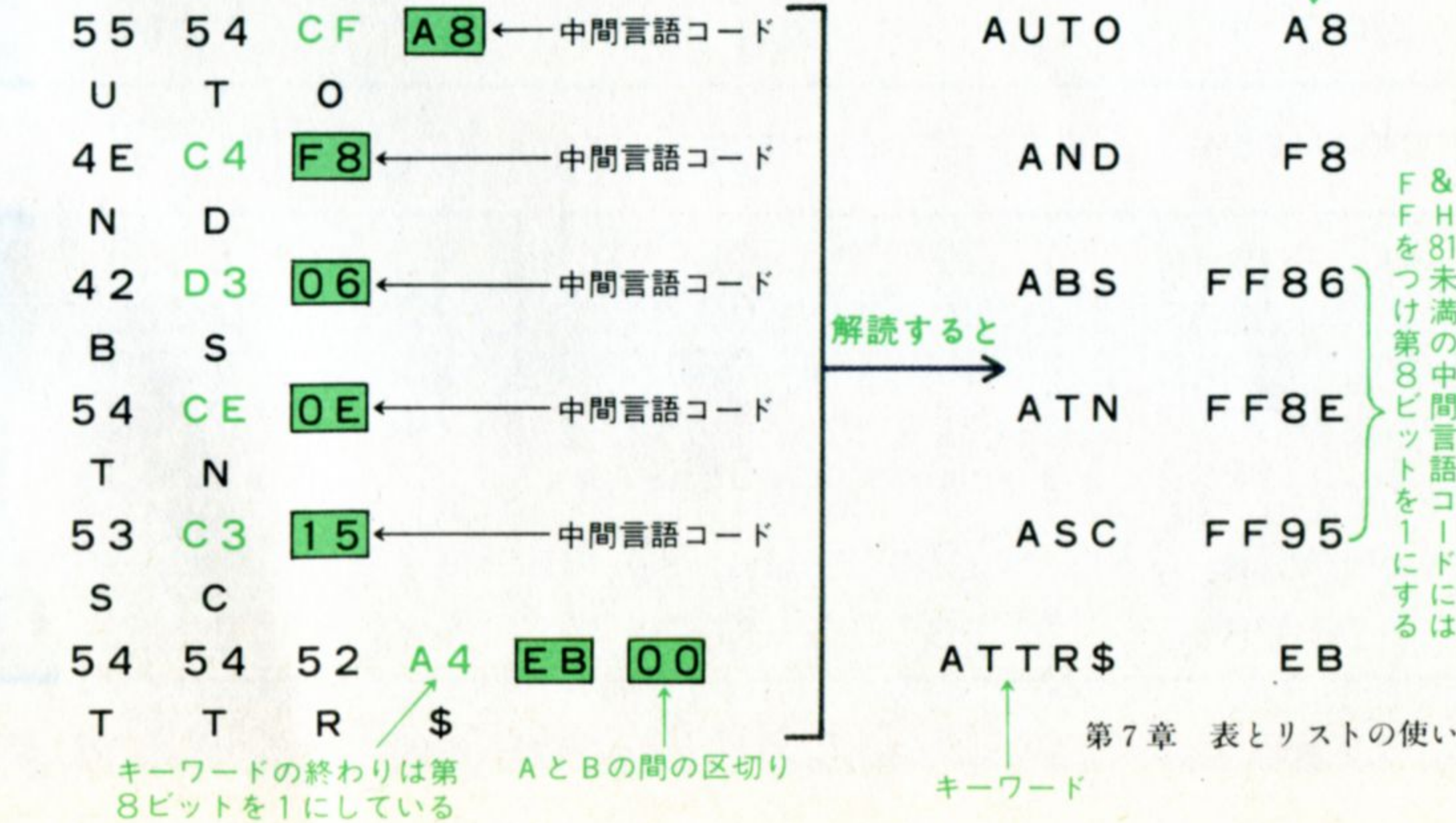
それではキーワードの間にある（やｘや␣やｋは何でしょうか。これは中間言語が化けているだけです。ＡとＢの境はどうやって明らかにしているのでしょう。それはＡＳＣＩＩ７ビットコードではわかりにくく、８ビットコードにもどるとわかりやすくなります。

もう一度先頭から書き出してみましょう。

					解読すると	キーワード	中間言語
55 U	54 T	CF O	A8 ←中間言語コード			AUTO	A8
4E N	C4 D	F8 ←中間言語コード				AND	F8
42 B	D3 S	06 ←中間言語コード				ABS	FF86
54 T	CE N	0E ←中間言語コード				ATN	FF8E
53 S	C3 C	15 ←中間言語コード				ASC	FF95
54 T	54 T	52 R	A4 $	EB	00	ATTR$	EB

キーワードの終わりは第８ビットを１にしている

ＡとＢの間の区切り

&Ｈ81未満の中間言語コードにはＦＦをつけ第８ビットを１にする

キーワードと中間言語コードの対応の仕組みがわかるでしょう。これを一々人間がやっていたのではたまりませんから、ＢＡＳＩＣで簡単なプログラムを作り、実行させることにします。プログラムは大変やさしいものです。これだけ調べておけばＢＡＳＩＣの作り方の構造がはっきりしてきます。たとえば、ＳＩＮというＢＡＳＩＣのキーワードに対する中間言語コードはＦＦ８９で、対応する機械語処理ルーチンは２Ｆ９１にあります。ですからＳＩＮ関数を利用したいときには２Ｆ９１からの機械語処理ルーチンを呼べばよいのです。すべてのルーチンが、このようにして利用可能になります。もう少しつっ込むにはブロック転送ルーチンや加算減算・乗算除算ルーチンの次元まで分解して考えねばなりませんが、それはもう少し勉強するか、より専門的な本を読まれるとよいと思います。

プログラム１

```
100 C=&H81
110 A$=" "
120 FOR AD=&H69FE TO &H6B00 STEP 2
130 IF AD=&H6AB0 THEN C=&H81:A$="FF"
140 IF AD=&H6B02 THEN C=&HD0
150 S=PEEK(AD)+PEEK(AD+1)*256
160 LPRINT A$;HEX$(C);TAB(10);RIGHT$("000"+HEX$(S),4)
170 C=C+1
180 NEXT
```

プログラム２

```
100 C=&H81
110 A$="FF"
115 C=&HD0
120 FOR AD=&H6B02 TO &H6B54 STEP 4
150 S=PEEK(AD)+PEEK(AD+1)*256
160 LPRINT A$;HEX$(C);TAB(10);RIGHT$("000"+HEX$(S),4);" ";
162 S=PEEK(AD+2)+PEEK(AD+3)*256
164 LPRINT RIGHT$("000"+HEX$(S),4)
170 C=C+1
180 NEXT
```

●中間言語コードとベーシックのキーワードの対応表

81	50E5	89	0BF9	91	0E54
82	08BF	8A	0B7C	92	522E
83	52BD	8B	0E05	93	18D9
84	0C77	8C	50A5	94	77DD
85	102D	8D	0BBF	95	0D01
86	5AC5	8E	0C41	96	1800
87	10F9	8F	0C79	97	15D7
88	0C9C	90	50CA	98	1B84

99	5140
9A	17FA
9B	0E4C
9C	18D4
9D	7071
9E	181A
9F	0C79
A0	5158
A1	5159
A2	515E
A3	519C
A4	657B
A5	0DCA
A6	0D8D
A7	1B40
A8	0DD5
A9	6F0E
AA	0AC4
AB	0AC7
AC	0ACA
AD	0ACD
AE	0FAA
AF	EE80
B0	EE83
B1	EE89
B2	0000
B3	0000
B4	0000
B5	EE86
B6	EEC5
B7	EE7A
B8	1C89
B9	1CD1
BA	EE98
BB	4798
BC	4A5C
BD	7198
BE	71A6
BF	EE8C
C0	4B04
C1	4854
C2	4855
C3	EE9B

C4	EE8F
C5	EE92
C6	49AB
C7	49AA
C8	48A3
C9	EE9E
CA	E826
CB	6EC6
CC	6ECE
CD	6EA6
CE	71B5
CF	6E9A
D0	6E96
D1	6EDA
D2	7367
D3	6EDE
D4	EEBF
D5	EEC2
D6	714E
D7	3EB4
D8	6ECA
D9	72AB

FF81	575A
FF82	578A
FF83	5793
FF84	20B3
FF85	2295
FF86	20A0
FF87	2E05
FF88	2F1A
FF89	2F91
FF8A	1F10
FF8B	2E6E
FF8C	2F8B
FF8D	302C
FF8E	EEC8
FF8F	58E4
FF90	17E5
FF91	1586
FF92	56F8
FF93	54CB
FF94	57B4

FF95	5704
FF96	5714
FF97	1B7A
FF98	5741
FF99	54C1
FF9A	54C6
FF9B	1581
FF9C	21A0
FF9D	2214
FF9E	223E
FF9F	2286
FFA0	4ABA
FFA1	4ABD
FFA2	4AC0
FFA3	4C51
FFA4	4C2F
FFA5	4C40
FFA6	4C62
FFA7	4AA1
FFA8	4AA4
FFA9	4AA7

FFD0	EE77	0000
FFD1	6EE2	6EBA
FFD2	6ED2	6ED6
FFD3	6EC2	6EB2
FFD4	3F31	0000
FFD5	6E9E	0000
FFD6	EE7D	0000
FFD7	0393	7F30
FFD8	6EF2	72C8
FFD9	6F02	721C
FFDA	0393	7D71
FFDB	0393	6EFA
FFDC	6EFE	7279
FFDD	0000	EEA1
FFDE	0000	EEA4
FFDF	0000	EEA7
FFE0	0000	EEAA
FFE1	EEB9	0000
FFE2	0000	EEAD
FFE3	EEB3	EEB0
FFE4	0000	EEB6

●ジャンプテーブルを作る

プログラム3

```
100 I=&H6B56
110 K=1
120 '
130 IF K>26 THEN END
140 '
150 S2=PEEK(I+1)
160 A$=HEX$(S2)
170 IF LEN(A$)<2 THEN A$="0"+A$
180 IF S2<32 THEN C$=" ":GOTO 220
190 IF (S2 AND 127)<32 THEN C$=" ":GOTO 220
200 C$=CHR$(S2 AND 127)
210 '
220 S1=PEEK(I)
230 B$=HEX$(S1)
240 IF LEN(B$)<2 THEN B$="0"+B$
250 IF S1<32 THEN D$=" ":GOTO 290
260 IF (S1 AND 127)<32 THEN D$=" ":GOTO 290
270 D$=CHR$(S1 AND 127)
280 '
290 LPRINT HEX$(I),CHR$(K+64),A$+B$
300 I=I+2:J=J+1:K=K+1
310 GOTO 130
```

6B56	A	6B8A
6B58	B	6BA0
6B5A	C	6BAF
6B5C	D	6C0E
6B5E	E	6C4A
6B60	F	6C6F
6B62	G	6C89
6B64	H	6C9B
6B66	I	6CA4
6B68	J	6CCE
6B6A	K	6CCF
6B6C	L	6CDC
6B6E	M	6D1C
6B70	N	6D40
6B72	O	6D4F
6B74	P	6D68
6B76	Q	6D97
6B78	R	6D98
6B7A	S	6DDA
6B7C	T	6E21
6B7E	U	6E41
6B80	V	6E4A
6B82	W	6E58
6B84	X	6E7B
6B86	Y	6E7F
6B88	Z	6E80

●ＢＡＳＩＣのキーワードと中間言語コードの対応表を作る

プログラム４

```
100 C=&H40
110 I=1
120 AD=&H6B56
130 A=PEEK(AD+1)*256+PEEK(AD)
140 PRINT CHR$(C+I);"        ";HEX$(A)
150 A$=CHR$(C+I)
160 N=PEEK(A)
170 IF N=0 THEN GOTO 300
180 N1=N AND 127
190 A$=A$+CHR$(N1)
200 IF N>128 THEN GOTO 230
210 A=A+1
220 GOTO 160
230 PRINT TAB(15);A$,
240 S=PEEK(A+1)
250 ICODE$=HEX$(S)
260 IF LEN(ICODE$)<2 THEN ICODE$="0"+ICODE$
270 IF S<128 THEN ICODE$="FF"+HEX$(S OR 128)
280 PRINT ICODE$
290 A=A+2:GOTO 150
300 I=I+1
310 IF I>26 THEN END
320 AD=AD+2:GOTO 130
```

A	6B8A		
		AUTO	A8
		AND	F8
		ABS	FF86
		ATN	FF8E
		ASC	FF95
		ATTR$	EB
B	6BA0		
		BSAVE	D5
		BLOAD	D4
		BEEP	D7
C	6BAF		
		CONSOLE	9D
		COPY	CD
		CLOSE	C0
		CONT	99
		CLEAR	92
		CSRLIN	FFD4
		CINT	FF9C
		CSNG	FF9D
		CDBL	FF9E
		CVI	FFA0
		CVS	FFA1
		CVD	FFA2
		COS	FF8C
		CHR$	FF96
		CALL	B1
		COMMON	B6
		CHAIN	B7
		COM	FFDA
		CIRCLE	CC
		COLOR	CB
		CLS	CE
		CMD	FFE4
D	6C0E		
		DELETE	A7
		DATA	84
		DIM	86
		DEFSTR	AA
		DEFINT	AB
		DEFSNG	AC
		DEFDBL	AD

		DSKO$	BA
		DEF	97
		DSKI$	EC
		DSKF	FFD0
		DATE$	FFD9
E	6C4A		
		ELSE	9F
		END	81
		ERASE	A3
		EDIT	A4
		ERROR	A5
		ERL	E4
		ERR	E5
		EXP	FF8B
		EOF	FFA3
		EQV	FB
F	6C6F		
		FOR	82
		FIELD	BC
		FILES	C3
		FN	E1
		FRE	FF8F
		FIX	FF9F
		FPOS	FFA6
G	6C89		
		GOTO	89
		GO TO	89
		GOSUB	8D
		GET	BD
H	6C9B		
		HEX$	FF9A
		HELP	D9
I	6CA4		
		INPUT	85
		ISET	FFE0
		IEEE	FFE1
		IRESET	FFE2
		IF	8B
		INSTR	E8
		INT	FF85
		INP	FF90
		IMP	FC
		INKEY$	EF
J	6CCE		
K	6CCF		
		KEY	FFDB
		KILL	C5
		KANJI	DB
L	6CDC		
		LOCATE	D6
		LPRINT	9B
		LLIST	9C
		LPOS	FF9B
		LET	88
		LINE	AE
		LOAD	C1
		LSET	C6
		LIST	93
		LFILES	C9
		LOG	FF8A
		LOC	FFA4
		LEN	FF92
		LEFT$	FF81
		LOF	FFA5
M	6D1C		
		MOTOR	FFD7
		MERGE	C2
		MOD	FD
		MKI$	FFA7
		MKS$	FFA8
		MKD$	FFA9
		MID$	FF83
		MON	CA
		MAP	FFD5
N	6D40		
		NEXT	83
		NAME	C4
		NEW	94
		NOT	E3
O	6D4F		
		OPEN	BB
		OUT	9A
		ON	95
		OR	F9
		OCT$	FF99
		OPTION	B8
		OFF	EE
P	6D68		
		PRINT	91
		PUT	BE
		POKE	98
		POLL	FFDF
		POS	FF91
		PEEK	FF97
		PSET	CF
		PRESET	D0
		POINT	FFD3
		PAINT	D1
		PEN	FFD8
Q	6D97		
R	6D98		
		RETURN	8E
		READ	87
		RUN	8A
		RESTORE	8C
		RBYTE	FFDE
		REM	8F
		RESUME	A6
		RSET	C7
		RIGHT$	FF82
		RND	FF88
		RENUM	A9
		RANDOMIZE	B9
		ROLL	D8

文字	アドレス	予約語	コード
S	6DDA		
		SCREEN	D3
		SEARCH	FFD6
		STOP	90
		SWAP	A2
		SET	BF
		SRQ	ED
		STATUS	FFE3
		SAVE	C8
		SPC(	E2
		STEP	DF
		SGN	FF84
		SQR	FF87
		SIN	FF89
		STR$	FF93
		STRING$	E6
		SPACE$	FF98
T	6E21		
		THEN	DD
		TRON	A0
		TROFF	A1
		TAB(	DE
		TO	DC
		TAN	FF8D
		TERM	D2
		TIME$	FFDC
U	6E41		
		USING	E7
		USR	E0
V	6E4A		
		VAL	FF94
		VIEW	FFD1
		VARPTR	EA
W	6E58		
		WIDTH	9E
		WINDOW	FFD2
		WAIT	96
		WHILE	AF
		WEND	B0
		WRITE	B5
		WBYTE	FFDD
X	6E7B		
		XOR	FA
Y	6E7F		
Z	6E80		

7・7 インデックスレジスタを使うこと

インテル8080

このあたりまで勉強してきますとインテル８０８０のアセンブリ言語の全体像が浮かんできたでしょう。ところで少し変った実験をしてみましょう。次のリストは＄Ｅ０００番地から＄Ｅ０１７番地までのメモリの内容を逆アセンブラにかけて見たものです。何のことやらさっぱりわかりませんね。？？？にいたっては首をひねらざるを得ません。ところがこのプログラムはちゃんと動作するので、ＧＥ０００として走らせると次ページに示すように、ある領域のデータを転送するプログラムであることがわかります。これはいわゆるブロック転送ルーチンであることがわかります。それではなぜ逆アセンブラにかからないのでしょうか。その秘密は、このプログラムがザイログＺ-８０用のアセンブラで書かれているからです。つまりインテル８０８０には用意されていない命令を使っているのでＰＣ-８８０１のモニタには理解できないのです。

ブロック転送ルーチン

ザイログZ-80

```
E000    DD          ???
E001    21 E021     LXI  H,E021
E004    FD          ???
E005    21 E031     LXI  H,E031
E008    0E 0A       MVI  C,0A
E00A    DD          ???
E00B    7E          MOV  A,M
E00C    00          NOP
E00D    FD          ???
E00E    77          MOV  M,A
E00F    00          NOP
E010    DD          ???
E011    23          INX  H
E012    FD          ???
E013    23          INX  H
E014    0D          DCR  C
E015    20 F3       JRNZ E00A
E017    76          HLT
```

ザイログのアセンブリ言語を使って書くと上のプログラムは次ページのようになります。いままでまったく登場して来なかった16ビットのインデックスレジスタＩＸ，ＩＹが使われていますし、相対ジャンプ命令のＪＲＮＺも使われています。

インデックスレジスタ

●ブロック転送前のメモリの様子

```
E020 00 01 01 01 01 01 01 01 01 01 01 00 00 00 00 00
E030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

●ブロック転送後のメモリの様子

```
E020 00 01 01 01 01 01 01 01 01 01 01 00 00 00 00 00
E030 00 01 01 01 01 01 01 01 01 01 01 00 00 00 00 00
```

●ブロック転送プログラム（Z-80の場合）

```
E000    DD 21 E0 21      LD IX,E021
E004    FD 21 E0 31      LD IY,E031
E008    0E 0A            LD C,0A
E00A    DD 7E 00         LD A,(IX)
E00D    FD 77 00         LD (IY),A
E010    DD 23            INX IX
E012    FD 23            INX IY
E014    0D               DEC C
E015    20 F3            JRNZ E00A
E017    76               HLT
```

これとまったく同じプログラムを８０８０のアセンブリ言語だけで書いてみますと下のようになり、見なれた形式がでてきます。実はザイログZ-８０は大変強力な命令をたくさん持ちあわせたマイクロプロセッサですが、８０８０のアセンブリ言語を使っているだけではその能力を十分に引き出すことはできません。

●ブロック転送プログラム（８０８０の場合）

```
E000    21 E021     LXI  H,E021
E003    11 E031     LXI  D,E031
E006    3E 0A       MVI  A,0A
E008    4F          MOV  C,A
E009    7E          MOV  A,M
E00A    12          STAX D
E00B    23          INX  H
E00C    13          INX  D
E00D    0D          DCR  C
E00E    C2 E009     JNZ  E009
E011    76          HLT
```

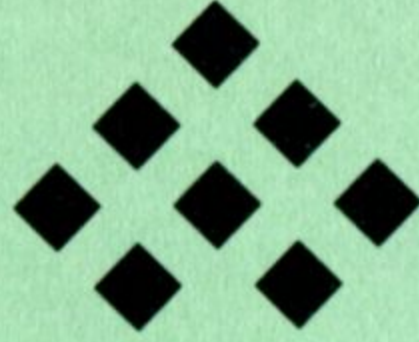
ＰＣ-８８０１のＢＡＳＩＣのＲＯＭを逆アセンブル出力させると読みにくい部分はあっても、それほど困難な部分はなく、Ｚ-８０的な書き方はしていても、８０８０系に非常に近いプログラミングの仕方をしているようです。特にインデックスレジスタＩＸ，ＩＹはあまり利用しないようで、これはプログラム作製者の癖か、何か確たる方針があってのことのようです。せっかくあるのに使わないのはもったいないような気もしますし、上手に使うとプログラムが楽になります。

JRNZ

なおＪＲＮＺは相対ジャンプ命令と言って８０８０用にはないものです。８０８０用のジャンプ命令はすべて絶対番地で指定する方式で、相対番地で指定するものはありません。ところがＰＣ-８８０１のモニタプログラムは、アセンブル作業の際にＪＲＮＺを使えるようにしてくれています。本来は相対番地の計算をするか、ラベルが使えないと使えないのですが、面白いことにＪＲＮＺは絶対番地指定で使えるようになっており、とても便利です。

第8章

サブルーチンの使い方

◆◆◆◆◆◆◆

8・1 BASICと機械語のプログラムをつなげる

DEF　USR文
USR関数

機械語とＢＡＳＩＣのプログラムをつなげるといっても、理屈は少しわかりにくい所がありますが、動作上は簡単です。まず一番やさしいものから始めましょう。前章でＢＡＳＩＣの機械語処理ルーチンの所在がわかっています。表をみますと、ＳＩＮは＄２Ｆ９１番地、ＣＯＳは＄２Ｆ８Ｂ番地です。ですからＳＩＮとＣＯＳを使おうとするときはＢＡＳＩＣプログラムの行番号110と120のようにＤＥＦ　ＵＳＲ文でＵＳＲ関数を定義しておく必要があります。ＰＣ-８８０１の場合、ＵＳＲ関数は０～９までの10個が使えます。つまりＢＡＳＩＣプログラムの中から最大10の機械語処理ルーチンが呼び出せます。０～９の番号の付け方は任意です。

もう１つ注意することは機械語処理ルーチンに引数をひきわたすやり方ですが、これはＢＡＳＩＣのほうで自動的に処理してくれるので簡単になっています。理屈を理解しないですませるにはＵＳＲ関数の引数に適当な値を設定してやればよいのです。もう少しわかりやすくいいますと、ＳＩＮ関数の機械語処理ルーチンへ飛んでもＳＩＮのどういう値を処理するのかわかりませんと、機械語処理ルーチンも困ってしまいます。ＳＩＮ４０°が欲しいと主張しないと何を計算すべきかがわかりません。この４０°に対応する部分を引数といいます。

引数

ＳＩＮ関数、ＣＯＳ関数では引数はラジアンで与えなければならないので、ＢＡＳＩＣプログラムの行番号150では角度をラジアンに変換しています。行番号160から行番号210までは結果をきれいに打ち出すための飾りにすぎません。

```
100 CLS
110 DEF USR0=&H2F91:'SIN
120 DEF USR1=&H2F8B:'COS
130 '
140 FOR I=0 TO 90 STEP 10
150 A=3.14159*I/180
160 LPRINT "SIN(";:LPRINT USING "##";I;
170 LPRINT ")=";
180 LPRINT USING "#.######";USR0(A),
190 LPRINT"    ";"COS(";:LPRINT USING "##";I;
200 LPRINT ")=";
210 LPRINT USING "#.######";USR1(A)
220 NEXT I
230 END
```

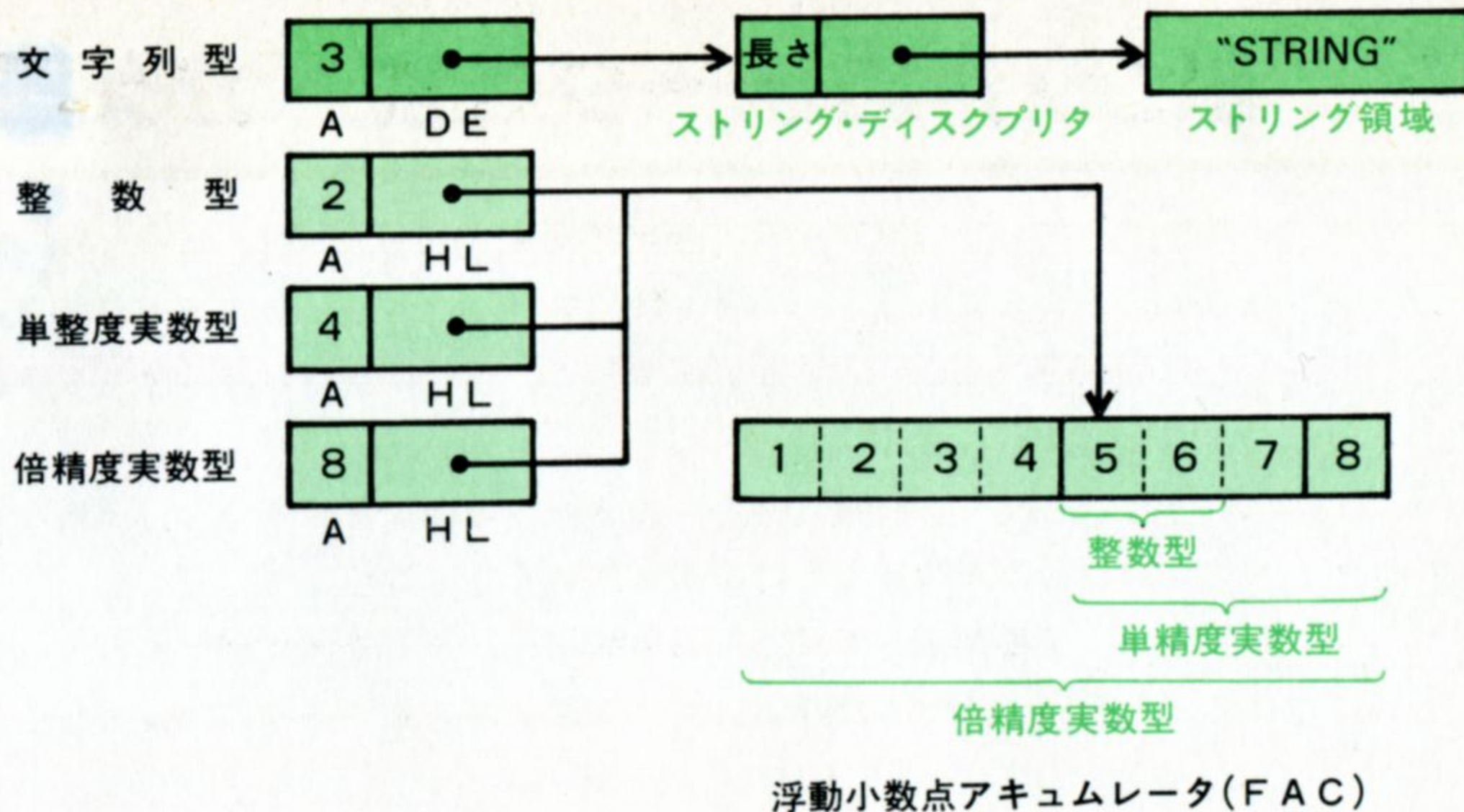
文字列型 | 3 | ● → 長さ | ● → "STRING"
A DE ストリング・ディスクプリタ ストリング領域

整数型 | 2 | ●
A HL

単整度実数型 | 4 | ●
A HL

倍精度実数型 | 8 | ●
A HL

1	2	3	4	5	6	7	8

整数型（5・6）
単精度実数型（5〜8）
倍精度実数型（1〜8）

浮動小数点アキュムレータ（FAC）

ＵＳＲ関数で引数の受けわたしは次のようにして行なわれます。まずＵＳＲ関数で使われている引数の型をＵＳＲ関数の処理ルーチンが判別してくれます。引数の型は４つあって文字列型、整数型、単精度実数型、倍精度実数型です。どの型であるかによって、ＤＥレジスタやＨＬレジスタに値が書き込まれます。文字列型以外の場合は引数の値は浮動小数点アキュムレータ（ＦＡＣ）に格納され、この値によって機械語処理ルーチンが処理を始めることになります。

浮動小数点アキュムレータ

処理が終わると結果の値は浮動小数点アキュムレータ（ＦＡＣ）に格納され、ＢＡＳＩＣにもどります。自分で作った機械語ルーチン以外のＰＣ－８８０１の機械語処理ルーチンでは、これらのことは何も理解しなくともすべてＢＡＳＩＣのほうで処理してくれますので簡単なのです。

前のプログラムと同じものをＢＡＳＩＣだけで組んだものを下図に示しておきます。

```
100 CLS
110 '
120 FOR I=0 TO 90 STEP 10
130 A=3.14159*I/180
140 LPRINT "SIN(";:LPRINT USING "##";I;
150 LPRINT ")=";
160 LPRINT USING "#.######";SIN(A),
170 LPRINT"     ";"COS(";:LPRINT USING "##";I;
180 LPRINT ")=";
190 LPRINT USING "#.######";COS(A)
200 NEXT I
210 END
```

8・2　16ビットのデータを並べかえる

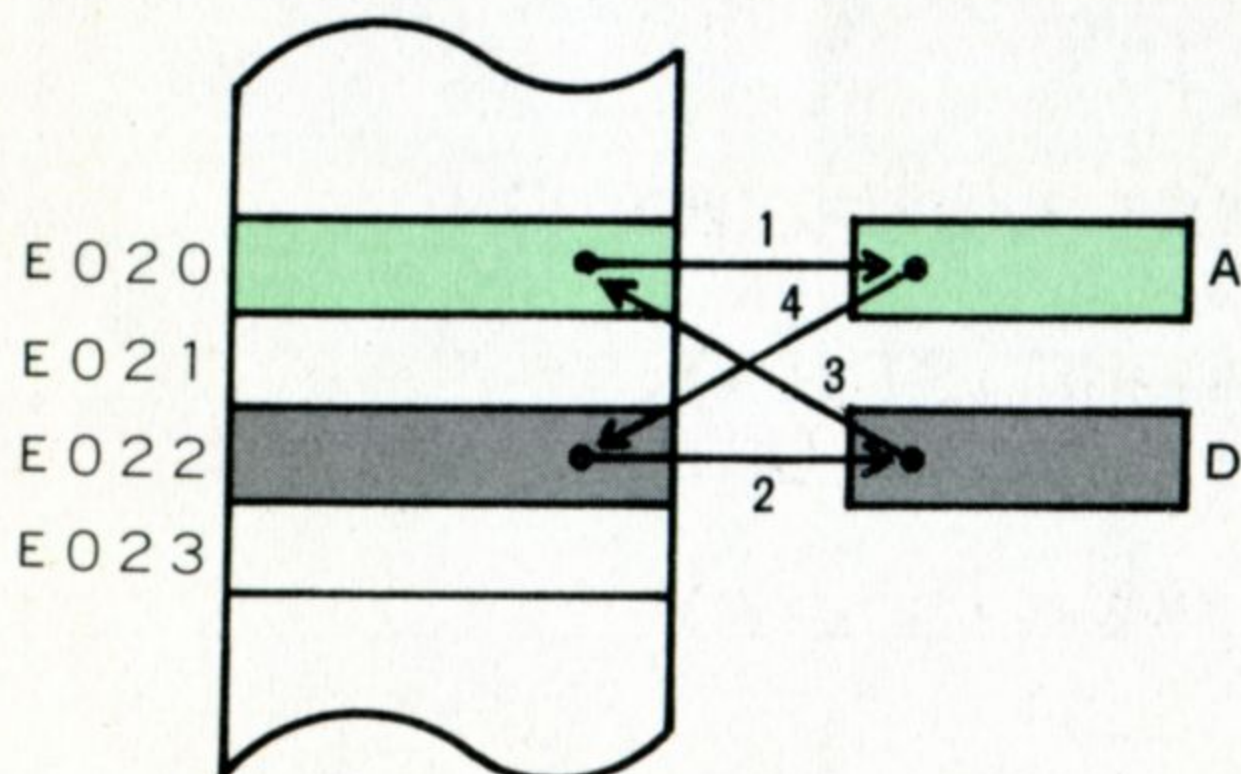
さて、機械語のプログラムをサブルーチンとして使うことを考えてみることにしましょう。あまり短いプログラムではBASICでやったほうが早いという苦情がでますので、少しは長さのあるプログラムを考えることにします。

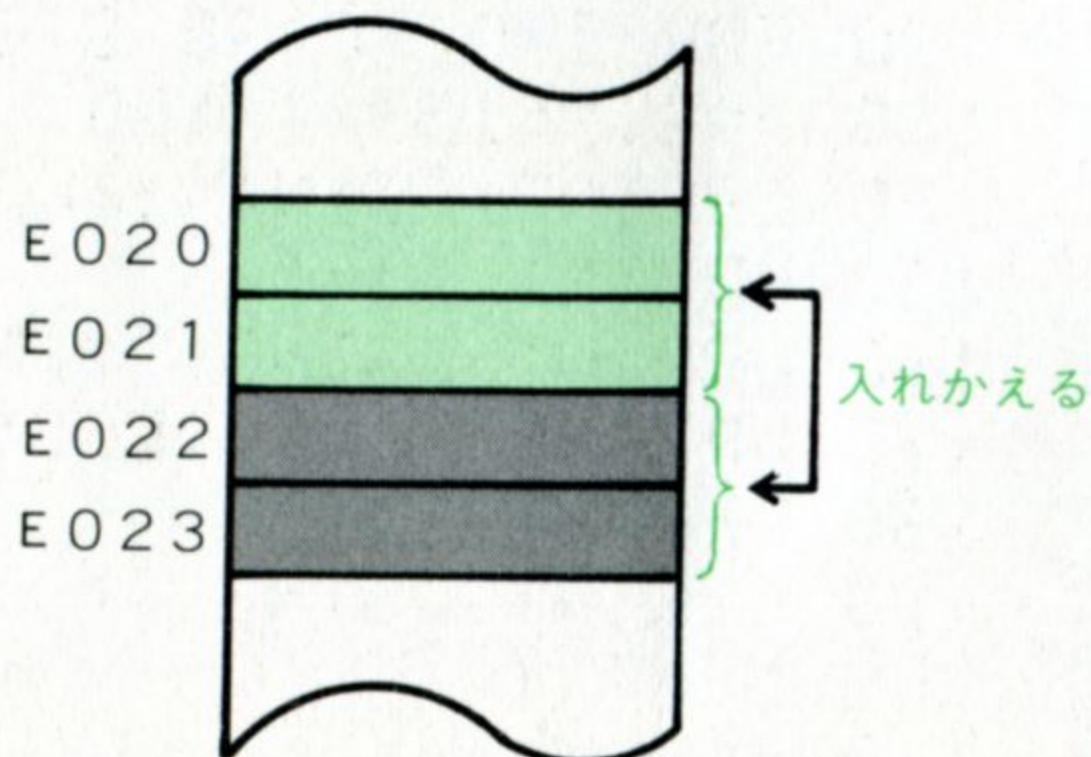
とはいっても、一挙に長いものを作るのは大変ですから、簡単なものからだんだん複雑なものを狙っていくことにします。

まず、1番地おいたメモリの中味を入れかえるプログラムを作ってみました。このプログラムはちょうど×をえがくようにクロスして入れかえるのです。

●メモリの中味を1番地おいて入れかえるプログラム

```
E000    21 E020     LXI   H,E020
E003    7E          MOV   A,M
E004    23          INX   H
E005    23          INX   H
E006    56          MOV   D,M
E007    77          MOV   M,A
E008    2B          DCX   H
E009    2B          DCX   H
E00A    72          MOV   M,D
E00B    76          HLT
```

```
E020 02 00 01 00     ←── プログラム実行前
E020 01 00 02 00     ←── プログラム実行後
```


1番地おいたメモリの中味を入れかえる

連続した2バイトを入れかえる

このプログラムをもう少し変更したのが次のプログラムです。これは入れかえ操作を２回行ないます。連続した２バイトをそっくりそのまま入れかえてしまうのです。

よく見るとわかりますが、同じブロックが２つあります。これをサブルーチン化することも考えてみましたが、実際には１バイト程度しか短くならないのでこのまま示しました。

●連続した２バイトを入れかえるプログラム

```
E000   21 E020    LXI  H,E020
E003   7E         MOV  A,M  ┐
E004   23         INX  H    │
E005   23         INX  H    │
E006   56         MOV  D,M  ├─まず、アキュムレータとDレジスタ
E007   77         MOV  M,A  │  を使って＄Ｅ０２０番地と＄Ｅ０２
E008   2B         DCX  H    │  ２番地の中味を入れかえる
E009   2B         DCX  H    │
E00A   72         MOV  M,D  ┘
E00B   23         INX  H    ┐
E00C   7E         MOV  A,M  │
E00D   23         INX  H    │
E00E   23         INX  H    ├─次に同じように＄Ｅ０２１番地と＄
E00F   56         MOV  D,M  │  Ｅ０２３番地の中味を入れかえる
E010   77         MOV  M,A  │
E011   2B         DCX  H    │
E012   2B         DCX  H    │
E013   72         MOV  M,D  ┘
E014   76         HLT
```

```
E020 03 04 01 02    ←── プログラム実行前
E020 01 02 03 04    ←── プログラム実行後
```

上のプログラムを変更して連続した２バイトの16ビット数を並べかえることを考えてみましょう。通常のように下位バイトが先行し上位バイトが続くものとします。

考え方は２バイトの16ビット数を比較し、小さい順に並んでいればそのままとし、並んでいなければ並べかえることにします。

いろいろな作り方が考えられますが、２バイトの16ビット数を引き算して大きさを比較し、その結果のキャリーフラグＣを利用して場合

わけをすることにします。何のためにこういうことをするかといいますと、漢字の並べかえをするプログラムを作ろうとしているのです。皆さんもよくご存知のことと思いますが、漢字は２バイトで表現されるのです（ただし厳密にいいますとＰＣ-８８０１の内部では扱い方が違います)。実用にするにはデータ構造の考え方を使ったり、もっと洗練されたプログラムを作る必要がありますが、特別な技巧を使わないで、どれだけのことができるかをためしてみるのも面白いことと思います。プログラムの実行結果をみますと無事に２バイトずつ入れかわっています。

●連続した２バイトの16ビット数を並べかえるプログラム

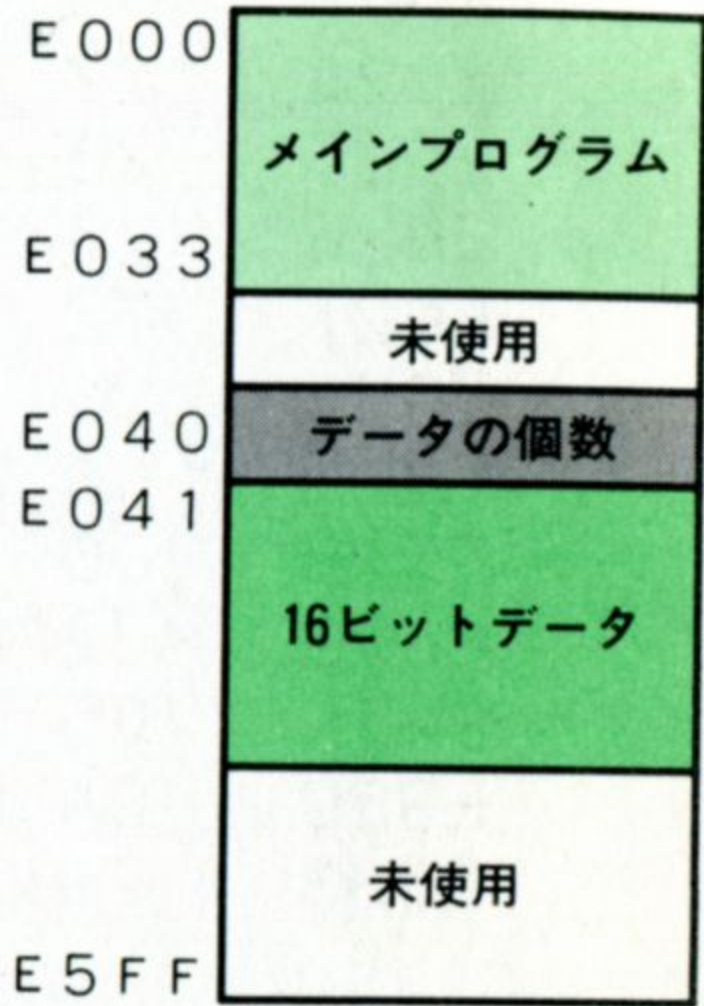
```
E000   21 E030   LXI  H,E030 ┐
E003   7E        MOV  A,M    │
E004   23        INX  H      ├─ はじめの16ビット数の下位バイトから次の16ビット数の上位バイトを引く(ここではキャリーフラグを問題とする)
E005   23        INX  H      │
E006   96        SUB  M      ┘
E007   2B        DCX  H      ┐
E008   7E        MOV  A,M    │
E009   23        INX  H      ├─ 次にはじめの16ビット数の上位バイトから次の16ビット数の上位バイトをキャリーも含めて引く
E00A   23        INX  H      │
E00B   9E        SBB  M      ┘
E00C   DA E023   JC   E023   ←─ 前の演算でキャリーがたてば、はじめの16ビット数のほうが小さいのだから終わり
E00F   2B        DCX  H      ┐
E010   2B        DCX  H      │
E011   2B        DCX  H      │
E012   7E        MOV  A,M    │
E013   23        INX  H      │
E014   23        INX  H      ├─ 下位バイト同士を入れかえる
E015   56        MOV  D,M    │
E016   77        MOV  M,A    │
E017   2B        DCX  H      │
E018   2B        DCX  H      │
E019   72        MOV  M,D    ┘
E01A   23        INX  H      ┐
E01B   7E        MOV  A,M    │
E01C   23        INX  H      │
E01D   23        INX  H      │
E01E   56        MOV  D,M    ├─ 上位バイト同士を入れかえる
E01F   77        MOV  M,A    │
E020   2B        DCX  H      │
E021   2B        DCX  H      │
E022   72        MOV  M,D    ┘
E023   76        HLT
```

```
E030 03 04 01 02 00 00 00 00
```
←プログラム実行前

```
E030 01 02 03 04 00 00 00 00
```
←プログラム実行後

さて次に16ビットソートのプログラムを作ってみました。８ビットソートのプログラムと考え方は同じです。 ここでのプログラムは256個の16ビットデータの並べかえまでできます。

だんだん作り上げてきましたのでプログラムの中味はわかるでしょう。ここのプログラムでは＄Ｅ０４０番地にデータの個数をしまい、16ビットデータは＄Ｅ０４１番地から並んでいます。プログラムの実行結果をみますと３個の16ビットデータが小さい順に並べかえられていることがわかります。

アドレス	内容
E000	メインプログラム
E033	未使用
E040	データの個数
E041	16ビットデータ
	未使用
E5FF	

●16ビットデータのソーティング(並べかえ)のプログラム

```
E000   06 00      MVI  B,00    ← Bレジスタをクリア(ソート終了がどうかのチェックに使う)
E002   21 E040    LXI  H,E040
E005   4E         MOV  C,M     ← データ数をCレジスタに入れる
E006   0D         DCR  C
E007   23         INX  H
E008   7E         MOV  A,M   ┐
E009   23         INX  H     │
E00A   23         INX  H     │
E00B   96         SUB  M     │
E00C   2B         DCX  H     ├ 2つの16ビット数の大小比較
E00D   7E         MOV  A,M   │
E00E   23         INX  H     │
E00F   23         INX  H     │
E010   9E         SBB  M     ┘
E011   2B         DCX  H
E012   DA E02B    JC   E02B    ← キャリーがたてば、つまり2数が小さい順になっていれば並べかえの必要がないからジャンプ
E015   2B         DCX  H     ┐
E016   2B         DCX  H     │
E017   7E         MOV  A,M   │
E018   23         INX  H     │
E019   23         INX  H     ├ 下位バイトの並べかえ
E01A   56         MOV  D,M   │
E01B   77         MOV  M,A   │
E01C   2B         DCX  H     │
E01D   2B         DCX  H     │
E01E   72         MOV  M,D   ┘
```

```
E01F    23           INX   H     ┐
E020    7E           MOV   A,M   │
E021    23           INX   H     │
E022    23           INX   H     │
E023    56           MOV   D,M   ├── 上位バイトの並べかえ
E024    77           MOV   M,A   │
E025    2B           DCX   H     │
E026    2B           DCX   H     │
E027    72           MOV   M,D   ┘
E028    23           INX   H
E029    06 01        MVI   B,01  ←── 並べかえありのマーク「01」をBレジスタに
E02B    0D           DCR   C     ←── データ数のカウンタをマイナス1
E02C    C2 E008      JNZ   E008  ←── カウンタが0でなければ＄Ｅ００８番地からのループへ
E02F    05           DCR   B
E030    CA E000      JZ    E000  ←── ソートが終了してなければ＄Ｅ００８番地からのループへ
E033    76           HLT
```

データ数

E040 03 56 78 34 56 12 34 ← プログラム実行前

16ビットデータ

E040 03 12 34 34 56 56 78 ← プログラム実行後

さて、プログラムができたので、これを使って楽しんでみましょう。まずＢＡＳＩＣプログラムでメモリの中味をのぞいて漢字で表示するプログラムとかみあわせることにします。プログラム実行前のメモリの中味は16進数で表示します。きわめて味気ないものですが、漢字に直して表示しますと面白いものがでてきます。この漢字の列は私の名前を読み込んだ漢詩をある偉い先生が作ってくださったものの一部です。大変気に入っていますので使わせていただきました。この漢詩の中に出てくる漢字をＪＩＳコードの表に従って並べ直すとどうなるかを次に示します。だいたい音読み順に並んでいるでしょう。

エイ・キ・キュウ・コウ・ショウ・セイ・ゼツ・ホウ・ユウ・リョウ

なかなか面白いものですね。子供の名前を考えるときにいろいろ漢字を並べかえて苦労したことを思い出します。本当の目的は電算写植機と日本語ワープロをつないで動かす時の漢字コードの変換のためですが、いまはこれ以上ふれないでおきましょう。

●メモリの中味を見るＢＡＳＩＣプログラム

```
100 CLS 3
110 M=PEEK(&HE040)
120 FOR I=1 TO M
130 N1=PEEK(&HE040+2*I-1)
140 N2=PEEK(&HE040+2*I)
150 N=N2*256+N1
160 PUT(I*20+100,60),KANJI(N),PSET,7,0
170 NEXT I
180 LOCATE 0,15
190 END
```

●プログラム実行前のメモリの様子

```
E040 0A 51 31 3A 4D 7B 34 64 40 57 35 24 40 2D 3E 30
E050 4B 51 39 42 4E
```

●プログラム実行後のメモルの様子

```
E040 0A 51 31 7B 34 57 35 51 39 2D 3E 24 40 64 40 30
E050 4B 3A 4D 42 4E
```

英雄既絶久世将飽育梁 ←―フログラム実行前の漢字データ

英既久育将世絶飽雄梁 ←―フログラム実行後の漢字データ

さて機械語のプログラムとＢＡＳＩＣのプログラムを連結してやることにします。最初はＤＥＦ　ＵＳＲ文を使います。このときにはパラメータはダミーで何の役目も果たしていません。

```
100 CLEAR ,&HDFFF
110 DEF USR0=&HE000
120 CLS 3
130 M=PEEK(&HE040)
140 FOR I=1 TO M
150 N1=PEEK(&HE040+2*I-1)
160 N2=PEEK(&HE040+2*I)
170 N=N2*256+N1
180 PUT(I*20+100,60),KANJI(N),PSET,7,0
190 NEXT I
```

```
200 '
210 B=USR0(A)
220 '
230 M=PEEK(&HE040)
240 FOR I=1 TO M
250 N1=PEEK(&HE040+2*I-1)
260 N2=PEEK(&HE040+2*I)
270 N=N2*256+N1
280 PUT(I*20+100,100),KANJI(N),PSET,7,0
290 NEXT I
300 END
```

ＤＥＦ　ＵＳＲ文の代りにＣＡＬＬ文を使うこともできます。この方が私は使いやすいと思います。この場合もパラメータはダミーです。

```
100 CLEAR ,&HDFFF
110 SORT=&HE000
120 CLS 3
130 M=PEEK(&HE040)
140 FOR I=1 TO M
150 N1=PEEK(&HE040+2*I-1)
160 N2=PEEK(&HE040+2*I)
170 N=N2*256+N1
180 PUT(I*20+100,60),KANJI(N),PSET,7,0
190 NEXT I
200 '
210 CALL SORT(A)
220 '
230 M=PEEK(&HE040)
240 FOR I=1 TO M
250 N1=PEEK(&HE040+2*I-1)
260 N2=PEEK(&HE040+2*I)
270 N=N2*256+N1
280 PUT(I*20+100,100),KANJI(N),PSET,7,0
290 NEXT I
300 END
```

機械語プログラムは最後のＨＬＴをＲＥＴ(リターン)に変更する必要があります。こうしませんとＢＡＳＩＣに戻ってきません。

```
E033    C9        RET
```

本格的にやるにはデータの構造をキチンと考えたり、変換テーブルを作るために様々の技巧を使ったりします。そうした事柄は本書の射程には入っていませんが、皆さんは奮起してぜひ勉強してみてください。そうするとコンピュータが面白くて面白くてたまらなくなりますよ。

8・3　機械語プログラムの呼び出し方

さて16ビットの並べかえのプログラムを反省してみましょう。あれはいやに長くなってしまいました。長いだけでなく見にくかったですね。そこでサブルーチン化することを考えましょう。本質的な部分は16ビットの入れかえの部分だけでしたから、その部分だけをとり出してみましょう。

●機械語プログラムを呼び出すBASICプログラム

```
100 CHS=&HE000
110 CALL CHS(A)
120 END
```

```
100 DEF USR0=&HE000
110 B=USR0(A)
120 END
```

●16ビットデータの入れかえの機械語プログラム

```
E000    21 E030    LXI   H,E030
E003    7E         MOV   A,M
E004    23         INX   H
E005    23         INX   H
E006    56         MOV   D,M
E007    77         MOV   M,A
E008    2B         DCX   H
E009    2B         DCX   H
E00A    72         MOV   M,D
E00B    23         INX   H
E00C    7E         MOV   A,M
E00D    23         INX   H
E00E    23         INX   H
E00F    56         MOV   D,M
E010    77         MOV   M,A
E011    2B         DCX   H
E012    2B         DCX   H
E013    72         MOV   M,D
E014    C9         RET
```

まったく同じものが2つ入っている

プログラムをよく調べてみますと、全く同じ部分が2つ入っています。そこでこれをサブルーチン化してみましょう。長さは全体としてそれほど短くならないのですが、見やすく扱いやすくなります。見やすいというメリットのほかにプログラムを改造したいときに一部分の修正が全体に波及することはありません。プログラムが出来上がったらできる限りサブルーチン化するのがよいと思います。

機械語プログラムのサブルーチンを呼び出すにはいろいろなテクニックがあります。先ほどのプログラムではＨＬレジスタの値を ＬＸＩ Ｈ，Ｅ０３０で指定しており、データの格納場所が変化するごとにプログラムを２バイトだけ書き直すことになります。

書き直し方は同じなのですが、ＨＬレジスタの指し示す番地の中にデータの格納場所を示すという手もあります。これは面白いテクニックです。

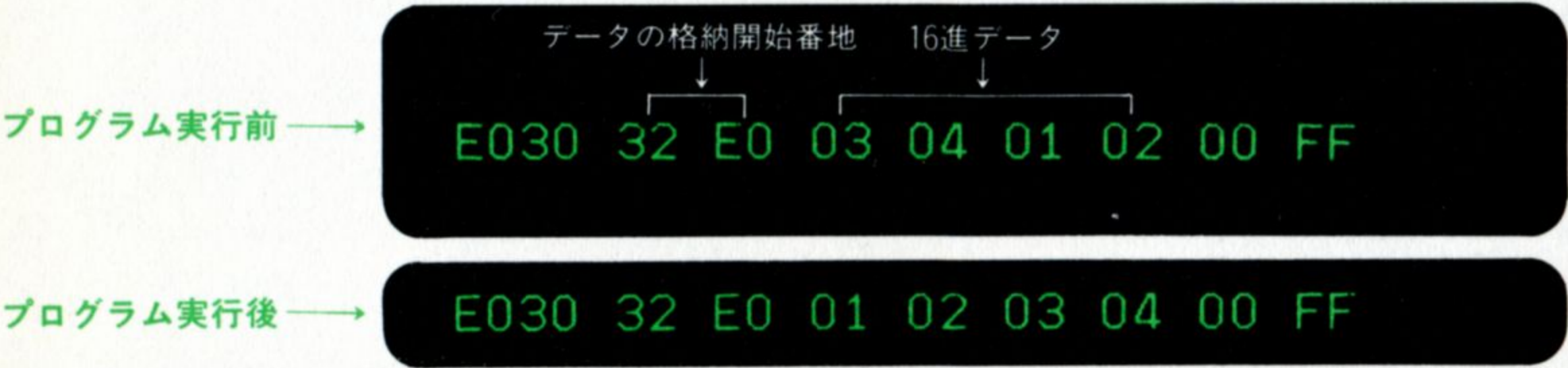
```
E000    21 E030       LXI   H,E030
E003    CD E010       CALL  E010
E006    23            INX   H
E007    CD E010       CALL  E010
E00A    C9            RET
E010    7E            MOV   A,M
E011    23            INX   H
E012    23            INX   H
E013    56            MOV   D,M
E014    77            MOV   M,A
E015    2B            DCX   H
E016    2B            DCX   H
E017    72            MOV   M,D
E018    C9            RET
```

呼び出している

サブルーチンプログラム

データの格納開始番地　16進データ

プログラム実行前 ──→ E030 32 E0 03 04 01 02 00 FF

プログラム実行後 ──→ E030 32 E0 01 02 03 04 00 FF

＄Ｅ０３０番地からのメモリの中味をごらんください。＄Ｅ０３０、＄Ｅ０３１番地に＄Ｅ０３２が入っているでしょう。これがポインタとなっており、データの格納場所がわかるのです。今の例題では＄Ｅ０３２番地は連続してでて来るので面白みがわかりませんが、よく味わっていただくと大変有用なテクニックであると思います。

モニタプログラムでアセンブリ言語プログラミングするのでない場合は、スタックポインタＳＰなどの設定をキチンとしなければならないのですが、モニタプログラムでプログラミングするときは、スタックポインタの設定を自分でしますとモニタプログラムのスタックポインタの設定と衝突して失敗します。

またプログラムの終りには今までＨＬＴを多用してきましたが、これからサブルーチンの呼び出しを何回も繰り返すようになると、プログラムの終りにはＲＥＴを使った方がよいと思います。この場合プログラムを走らせるためにＧＯ（ゴー）コマンドを使えなくなり、ＢＡＳＩＣプログラムで起動するとかの手を使うことになりますが、ＨＬＴを使ったままですと、プログラムが停止せずフロッピィが暴走したり、画面が消えたり、大変なことが起きる可能性があります。

もう一つ工夫してみたのが次のプログラムです。

●機械語プログラムを呼び出すＢＡＳＩＣプログラム

```
100 DEF USR0=&HE000
110 B=USR0(&HE030)
120 END
```

●機械語プログラム

```
E000   5E          MOV  E,M
E001   23          INX  H
E002   56          MOV  D,M
E003   EB          XCHG
E004   CD E010     CALL E010
E007   23          INX  H
E008   CD E010     CALL E010
E00B   C9          RET
E010   7E          MOV  A,M
E011   23          INX  H
E012   23          INX  H
E013   56          MOV  D,M
E014   77          MOV  M,A
E015   2B          DCX  H
E016   2B          DCX  H
E017   72          MOV  M,D
E018   C9          RET
```

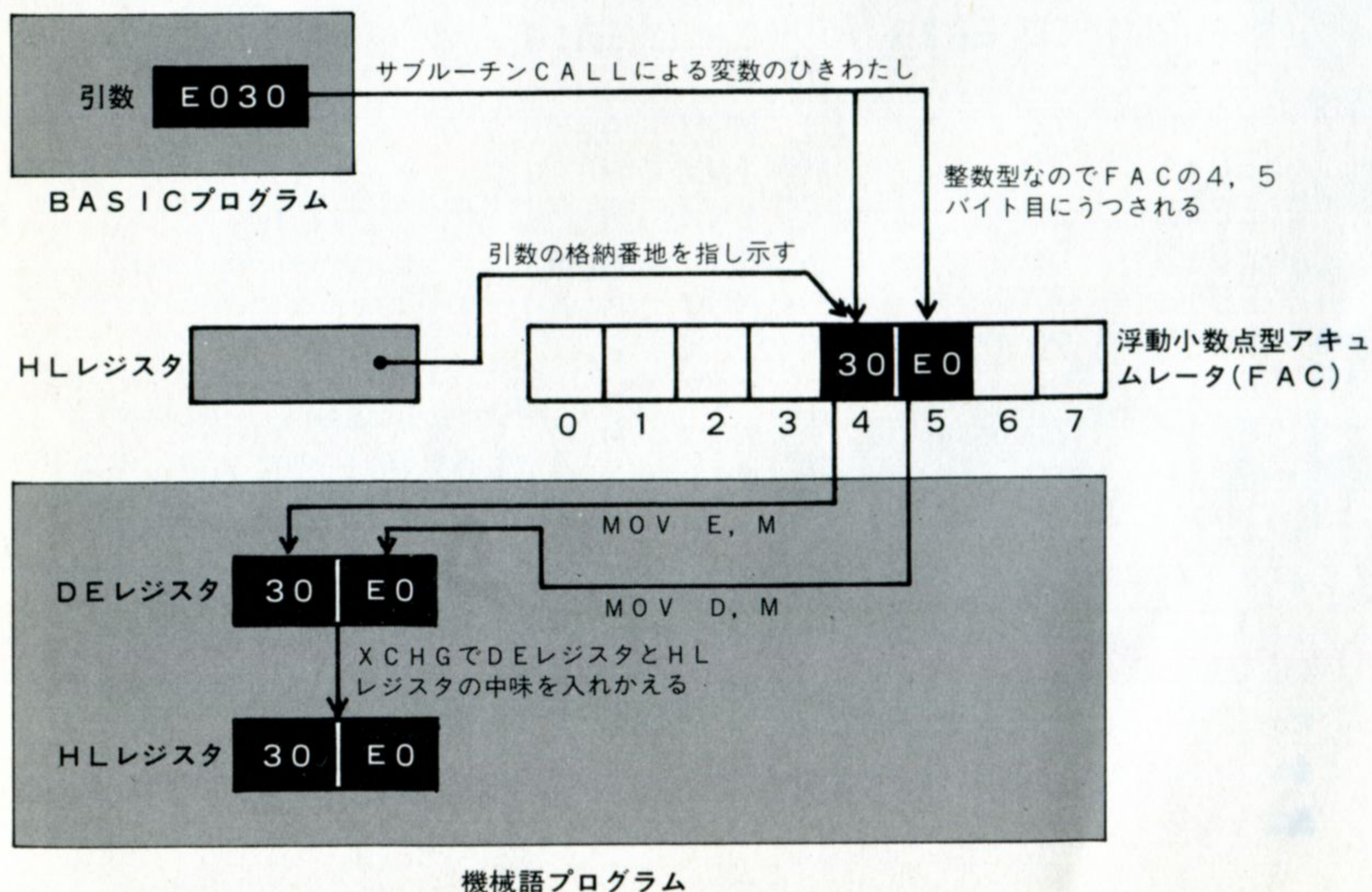
引数(パラメータ)

今まではＵＳＲ文の引数は完全なダミーで、何の役も果たしていませんでしたが、ここでのプログラムはＢＡＳＩＣプログラムと機械語プログラムの間で引数のひきわたしを行なっています。具体的にいいますと、引数にデータの最初の格納番地をわたしています。これは整数の２バイトになります。

従って引数は自動的に浮動小数点型のアキュムレータ（ＦＡＣ）の４、５バイト目にうつされます。浮動小数点型のアキュムレータはメモリの一部に仮想的に設けられるものです。それがどこにあるかについては特に意識する必要はありません。大切なのは整数型、実数型、倍精度型、文字型のときに浮動小数点型のアキュムレータの何バイト目から何バイト目に引数がどの順序にしまわれているかを知っておくことです。ＵＳＲ文を使い、機械語プログラムへとびますと、自動的にＨＬレジスタは、引数のしまわれている先頭番地を指し示してくれるからです。これはきわめて都合のよいことなのですが、機械語プログラムの中でＨＬレジスタを使う場合には注意が必要です。

つまり、機械語プログラムへとんだ時点ではＨＬレジスタは浮動小数点型のアキュムレータへのポインタとして使われていますので、これを適当なレジスタ対、たとえばＤＥレジスタに待避させてやり、それからＸＣＨＧなどで再びＨＬレジスタ対を使い始めるなどの工夫が必要です。

機械語プログラムの呼び出し方にはもっともっと凝ったやり方がたくさんあります。しかし大切なことはキチンと動作するプログラムを書くことであり、技巧に走って動作しないプログラムを書いたのでは仕方がありません。まず動くプログラムを確実な手法で書くことが一番です。

引数　E030
BASICプログラム
サブルーチンCALLによる変数のひきわたし
整数型なのでFACの4, 5
バイト目にうつされる
引数の格納番地を指し示す
HLレジスタ
30　E0
0　1　2　3　4　5　6　7
浮動小数点型アキュ
ムレータ(FAC)
MOV E, M
DEレジスタ　30　E0
MOV D, M
XCHGでDEレジスタとHL
レジスタの中味を入れかえる
HLレジスタ　30　E0
機械語プログラム

8・4　今後の勉強の仕方について

とうとう本文の最後になってしまいました。私の始めの考えでは、入出力（I／O）と割り込み（INTERRUPT）とプログラムの作り方についても話を展開する予定でしたが、ページもつきてしまいました。今後どういうふうに勉強したらよいかについて私なりの考えを述べておきましょう。

まず本書は、本文中で何度も強調しましたようにインテル８０８０系のアセンブリ言語だけで書かれています。ＰＣ-８８０１がザイログＺ８０を搭載していることを思いあわせると、ぜひＺ８０の勉強をしていただきたいと思います。本当は勉強をする際にはキチンとしたＺ８０用アセンブラと逆アセンブラを持っていたほうがよいのです。雑誌掲載の記事の中にもよいものはありますが、ある程度の投資ができれば次のようなすぐれたソフトウエアを揃えたほうがよいと思います。

❶ **DUAD-88D（機械語開発ツール、アスキー社）**

ただしフロッピィディスクを持っていることが前提となります。

Ｚ８０について書いた本はたくさんありすぎてどれをおすすめしてよいかわかりません。洋書であれば次の本がよいと思います。

❷ L. A. Leventhal "Z80 ASSEMBLY LANGUAGE PROGRAMMING" Osborne & McGraw-Hill

❸ R. Zaks Programming the "Z80" Sybex

文献❷は最近版型が１段大きくなり立派になりました。表紙もかわったようです。どちらかといえば８０８０的な書き方をしています。なお同じ L. A. Leventhalの

❹ L. A. Leventhal "8080 A／8085 ASSEMBLY LANGUAGE PROGRAMMING" Osborne & McGraw-Hill

は少し古くなった部分もありますが、名著で大変参考になりました。

Ｚ８０の勉強はソフトウエアだけでなく、ハードウエアの面からもされたほうがよいと思います。その点でトランジスタ技術誌に連載された、

❺ **相沢一石"作りながら学ぶマイコン設計トレーニング"「トランジスタ技術誌」1981年11月～1982年12月号（14回連載）**

は素晴しいものだと思います。研究室の学生に製作させましたが、大変勉強になったようです。システムが大きくなることを考えると、しっかりしたマザーバスと、ガラスエポキシの基板を使ったほうが後悔

がないように思います。

インテル系の８０８０用のアセンブリ言語を勉強することはＣＰ／Ｍというマイクロコンピュータのオペレーティング・システムへと勉強がすすむことになります。この目的のためには次の本がよいと思います。

❻ J. N. Fernandez, R. Ashley 中村和郎訳『ＣＰ／Ｍ演習』 工学図書

❼ 前田英明『ＣＰ／Ｍの使い方』 工学図書

❻は学生に課題で与えたことがありますが、１、２週間でマスターできて大変好評だったようです。❼もすぐれたよい本です。

なおアスキー出版からＣＰ／Ｍ３部作として

❽ 村瀬康治『入門ＣＰ／Ｍ』
村瀬康治『実習ＣＰ／Ｍ』
村瀬康治『応用ＣＰ／Ｍ』

がでています。大変具体的に書かれています。

ＰＣ-８８０１のＲＯＭの中味の機械語プログラムを理解するためには

❾ 川村清『ＰＣ-８８０１ N_{88}-ＢＡＳＩＣ 解析マニュアル』秀和システムトレーディング株式会社

がすぐれていて、非常に参考になります。ＰＣ-８８０１とＰＣ-８００１は兄弟なのでＰＣ-８０００シリーズのＲＯＭ解説書も参考になります。

❿ 川村清『ＰＣ-８００１ ＢＡＳＩＣ ＳＯＵＲＣＥ ＰＲＯＧＲＡＭ ＬＩＳＴＩＮＧＳ』 秀和システムトレーディング株式会社

⓫ 川村清『ＰＣ-８００１ マシン語活用ハンドブック 初級編——内部サブルーチンのすべて——』 秀和システムトレーディング株式会社

⓬ 牟田慎一郎他『ＰＣファミリーテクニカル・ノウハウ集 ＰＣ-８０００シリーズ編 ＰＣ Tech know ８０００』アスキー出版

⓭ 栗山浩一他『ＰＣファミリーテクニカル・ノウハウ集 ＰＣ-８８００編 ＰＣ Tech know ８８００』アスキー出版

⓮ J. Farvour "MICROSOFT BASIC DECORDED & OTHER MYSTERIES for the TRS８０" IJG Computer Service

これはシリーズ物でＴＲＳ８０のマイクロソフトＢＡＳＩＣを解析していますが、⓮が一番すぐれているようです。克明さにかけては❿から⓭に劣りますが、視点が独特な所があって参考になります。

⓯ 工藤丈彦他『ＰＣ-８８０１ グラフィックスのすべて』 アスキー出版

はグラフィックスの本ですが、内部の記述をよく調べてみますと、ふ

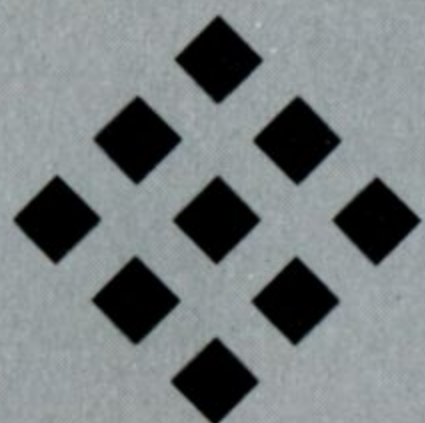
つうでは知ることのできないＩ／Ｏポートの番号やメモリの割当てを使っており、またグラフィックスの本としてもすぐれているのでぜひ御一読をおすすめいたします。

機械語プログラムが上手になるためには、やはりＰＣ-８８０１に向かって１つ１つ入力して失敗を重ねて行くほかにはありません。それにはまずＰＣ-８８０１用のマニュアルはすみずみまでよく読んでください。

⑯ NEC『PC-8801 USER'S MANUAL』
日本電気株式会社
NEC『PC-8801 N88-BASIC N-BASIC
REFERENCE MANUAL』 日本電気株式会社

なお、本書を草するにあたり、上記の文献類には大変お世話になりました。あらためて御礼申し上げます。

第9章

ふろく

ニーモニック→機械語　対照表

(MOV, INR, DCR, MVI)命令

r_2(r)	A	B	C	D	E	H	L	M
MOV A, r_2	7F	78	79	7A	7B	7C	7D	7E
MOV B, r_2	47	40	41	42	43	44	45	46
MOV C, r_2	4F	48	49	4A	4B	4C	4D	4E
MOV D, r_2	57	50	51	52	53	54	55	56
MOV E, r_2	5F	58	59	5A	5B	5C	5D	5E
MOV H, r_2	67	60	61	62	63	64	65	66
MOV L, r_2	6F	68	69	6A	6B	6C	6D	6E
MOV M, r	77	70	71	72	73	74	75	—
INR r	3C	04	0C	14	1C	24	2C	34
DCR r	3D	05	0D	15	1D	25	2D	35
MVI r, B_2	3E	06	0E	16	1E	26	2E	36

演算・論理(I)

r	A	B	C	D	E	H	L	M
ADD r	87	80	81	82	83	84	85	86
ADC r	8F	88	89	8A	8B	8C	8D	8E
SUB r	97	90	91	92	93	94	95	96
SBB r	9F	98	99	9A	9B	9C	9D	9E
ANA r	A7	A0	A1	A2	A3	A4	A5	A6
XRA r	AF	A8	A9	AA	AB	AC	AD	AE
ORA r	B7	B0	B1	B2	B3	B4	B5	B6
CMP r	BF	B8	B9	BA	BB	BC	BD	BE

(LXI, DAD, INX, DCX)命令

×	B	D	H	SP
LXI ×, B_3B_2	01	11	21	31
DAD ×	09	19	29	39
INX × DCX ×	03 0B	13 1B	23 2B	33 3B

演算・論理(II)

ADI B_2 ACI B_2	C6 CE
SUI B_2 SBI B_2	D6 DE
ANI B_2 XRI B_2 ORI B_2	E6 EE F6
CPI B_2	FE

スタック操作命令

×	B	D	H	PSW
PUSH× POP ×	C5 C1	D5 D1	E5 E1	F5 F1

Accとのデータ転送(8ビット)

	STAX B	LDAX B	STAX D	LDAX D	STA B_3B_2	LDA B_3B_2
オペレーション	〔(B)(C)〕⇆(A)		〔(D)(E)〕⇆(A)		〔⟨B_3⟩⟨B_2⟩〕⇆(A)	
機　械　語	02	0A	12	1A	32	3A

HLレジスタとのデータ転送(16ビット)

	SHLD B_3B_2	LHLD B_3B_2	XTHL	XCHG	PCHL	SPHL
オペレーション	〔⟨B_3⟩⟨B_2⟩+1〕 〔⟨B_3⟩⟨B_2⟩〕 ↑ (H)(L)	〔⟨B_3⟩⟨B_2⟩+1〕 〔⟨B_3⟩⟨B_2⟩〕 ↓ 〃	〔(SP)+1〕 〔(SP)〕 ↕ 〃	(D)(E) ↕ 〃	(PC) ↑ 〃	(SP) ↑ 〃
機　械　語	22	2A	E3	EB	E9	F9

分岐，コール，リターン命令

	······→	NZ	Z	NC	C	PO	PE	P	M
JMP B_3B_2 CALL B_3B_2 RET	C3 CD C9	C2 C4 C0	CA CC C8	D2 D4 D0	DA DC D8	E2 E4 E0	EA EC E8	F2 F4 F0	FA FC F8
条　　件	······→	Z=0, Z=1		C=0, C=1		P=0, P=1		S=0, S=1	

リスタート命令

× RST ×	0	1	2	3	4	5	6	7
RST ×	C 7	C F	D 7	D F	E 7	E F	F 7	F F

データ・バスとフラグ

DB_7	DB_6	DB_5	DB_4	DB_3	DB_2	DB_1	DB_0
S	Z	SUB	CY_4	"1"	P	"1"	C

回転・シフト命令

	RLC	RRC	RAL	RAR
オペレーション	$A_0 \leftarrow A_7$	$A_0 \rightarrow A_7$	$A_0 \leftarrow C$	$C \rightarrow A_7$
機　械　語	0 7	0 F	1 7	1 F

入出力・割込制御命令

	OUT B_2	IN B_2	D I	E I
機　械　語	D 3	D B	F 3	F B

Acc補正・C操作命令

	DAA	CMA	STC	CMC
オペレーション	補正	$A \leftarrow \overline{A}$	$C \leftarrow 1$	$C \leftarrow \overline{C}$
機　械　語	2 7	2 F	3 7	3 F

その他の命令

	HLT	NOP
機　械　語	76	00

機械語→ニーモニック対照表

16進	ニーモニック	16進	ニーモニック	16進	ニーモニック	16進	ニーモニック
00	NOP	40	MOV B, B	80	ADD B	C0	RNZ
01	LXI B, B_3 B_2	41	MOV B, C	81	ADD C	C1	POP B
02	STAX B	42	MOV B, D	82	ADD D	C2	JNZ B_3 B_2
03	INX B	43	MOV B, E	83	ADD E	C3	JMP B_3 B_2
04	INR B	44	MOV B, H	84	ADD H	C4	CNZ B_3 B_2
05	DCR B	45	MOV B, L	85	ADD L	C5	PUSH B
06	MVI B, B_2	46	MOV B, M	86	ADD M	C6	ADI B_2
07	RLC	47	MOV B, A	87	ADD A	C7	RST 0
08	—	48	MOV C, B	88	ADC B	C8	RZ
09	DAD B	49	MOV C, C	89	ADC C	C9	RET
0A	LDAX B	4A	MOV C, D	8A	ADC D	CA	JZ B_3 B_2
0B	DCX B	4B	MOV C, E	8B	ADC E	CB	—
0C	INR C	4C	MOV C, H	8C	ADC H	CC	CZ B_3 B_2
0D	DCR C	4D	MOV C, L	8D	ADC L	CD	CALL B_3 B_2
0E	MVI C, B_2	4E	MOV C, M	8E	ADC M	CE	ACI B_2
0F	RRC	4F	MOV C, A	8F	ADC A	CF	RST 1
10	—	50	MOV D, B	90	SUB B	D0	RNC
11	LXI D, B_3 B_2	51	MOV D, C	91	SUB C	D1	POP D
12	STAX D	52	MOV D, D	92	SUB D	D2	JNC B_3 B_2
13	INX D	53	MOV D, E	93	SUB E	D3	OUT B_2
14	INR D	54	MOV D, H	94	SUB H	D4	CNC B_3 B_2
15	DCR D	55	MOV D, L	95	SUB L	D5	PUSH D
16	MVI D, B_2	56	MOV D, M	96	SUB M	D6	SUI B_2
17	RAL	57	MOV D, A	97	SUB A	D7	RST 2
18	—	58	MOV E, B	98	SBB B	D8	RC
19	DAD D	59	MOV E, C	99	SBB C	D9	—
1A	LDAX D	5A	MOV E, D	9A	SBB D	DA	JC B_3 B_2
1B	DCX D	5B	MOV E, E	9B	SBB E	DB	IN B_2
1C	INR E	5C	MOV E, H	9C	SBB H	DC	CC B_3 B_2
1D	DCR E	5D	MOV E, L	9D	SBB L	DD	—
1E	MVI E, B_2	5E	MOV E, M	9E	SBB M	DE	SBI B_2
1F	RAR	5F	MOV E, A	9F	SBB A	DF	RST 3
20	RIM	60	MOV H, B	A0	ANA B	E0	RPO
21	LXI H, B_3 B_2	61	MOV H, C	A1	ANA C	E1	POP H
22	SHLD B_3 B_2	62	MOV H, D	A2	ANA D	E2	JPO B_3 B_2
23	INX H	63	MOV H, E	A3	ANA E	E3	XTHL
24	INR H	64	MOV H, H	A4	ANA H	E4	CPO B_3 B_2
25	DCR H	65	MOV H, L	A5	ANA L	E5	PUSH H
26	MVI H, B_2	66	MOV H, M	A6	ANA M	E6	ANI B_2
27	DAA	67	MOV H, A	A7	ANA A	E7	RST 4
28	—	68	MOV L, B	A8	XRA B	E8	RPE
29	DAD H	69	MOV L, C	A9	XRA C	E9	PCHL
2A	LHLD B_3 B_2	6A	MOV L, D	AA	XRA D	EA	JPE B_3 B_2
2B	DCX H	6B	MOV L, E	AB	XRA E	EB	XCHG
2C	INR L	6C	MOV L, H	AC	XRA H	EC	CPE B_3 B_2
2D	DCR L	6D	MOV L, L	AD	XRA L	ED	—
2E	MVI L, B_2	6E	MOV L, M	AE	XRA M	EE	XRI B_2
2F	CMA	6F	MOV L, A	AF	XRA A	EF	RST 5
30	SIM	70	MOV M, B	B0	ORA B	F0	RP
31	LXISP, B_3 B_2	71	MOV M, C	B1	ORA C	F1	POP PSW
32	STA B_3 B_2	72	MOV M, D	B2	ORA D	F2	JP B_3 B_2
33	INX SP	73	MOV M, E	B3	ORA E	F3	DI
34	INR M	74	MOV M, H	B4	ORA H	F4	CP B_3 B_2
35	DCR M	75	MOV M, L	B5	ORA L	F5	PUSH PSW
36	MVI M, B_2	76	HLT	B6	ORA M	F6	ORI B_2
37	STC	77	MOV M, A	B7	ORA A	F7	RST 6
38	—	78	MOV A, B	B8	CMP B	F8	RM
39	DAD SP	79	MOV A, C	B9	CMP C	F9	SPHL
3A	LDA B_3 B_2	7A	MOV A, D	BA	CMP D	FA	JM B_3 B_2
3B	DCX SP	7B	MOV A, E	BB	CMP E	FB	EI
3C	INR A	7C	MOV A, H	BC	CMP H	FC	CM B_3 B_2
3D	DCR A	7D	MOV A, L	BD	CMP L	FD	—
3E	MVI A, B_2	7E	MOV A, M	BE	CMP M	FE	CPI B_2
3F	CMC	7F	MOV A, A	BF	CMP A	FF	RST 7

命令一覧表

(1) 命令一覧表（アルファベット順）

命　令	名　　称
ACI	ADD WITH CARRY IMMEDIATE TO ACCUMULATOR
ADC	ADD REGISTER OR MEMORY WITH CARRY TO ACCUMULATOR
ADD	ADD REGISTER OR MEMORY TO ACCUMULATOR
ADI	ADD IMMEDIATE TO ACCUMULATOR
ANA	AND REGISTER OR MEMORY WITH ACCUMULATOR
ANI	AND IMMEDIATE WITH ACCUMULATOR
CALL	CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND
CC	CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND BUT ONLY IF THE CARRY STATUS EQUALS 1
CM	CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND BUT ONLY IF THE SIGN STATUS EQUALS 1
CMA	COMPLEMENT THE ACCUMULATOR
CMC	COMPLEMENT THE CARRY STATUS
CMP	COMPARE REGISTER OR MEMORY WITH ACCUMULATOR
CNC	CALL THE SUBROUTINE IDENTIFIED BY THE OPERAND BUT ONLY IF THE CARRY STATUS EQUALS 0
CNZ	CALL THE SUBROUTINE IDENTIFIED BY THE OPERAND BUT ONLY IF THE ZERO STATUS EQUALS 0
CP	CALL THE SUBROUTINE IDENTIFIED BY THE OPERAND BUT ONLY IF THE SIGN STATUS EQUALS 0
CPE	CALL THE SUBROUTINE IDENTIFIED BY THE OPERAND BUT ONLY IF THE PARITY STATUS EQUALS 1
CPI	COMPARE ACCUMULATOR CONTENTS WITH IMMEDIATE DATA
CPO	CALL THE SUBROUTINE IDENTIFIED BY THE OPERAND BUT ONLY IF THE PARITY STATUS EQUALS 0
CZ	CALL THE SUBROUTINE IDENTIFIED BY THE OPERAND BUT ONLY IF THE ZERO STATUS EQUALS 1
DAA	DECIMAL ADJUST ACCUMULATOR
DAD	ADD A REGISTER PAIR TO H AND L
DCR	DECREMENT REGISTER OR MEMORY CONTENTS
DCX	DECREMENT REGISTER PAIR
DI	DISABLE INTERRUPTS
EI	ENABLE INTERRUPTS
HLT	HALT

命令	名称
IN	INPUT TO ACCUMULATOR
INR	INCREMENT REGISTER OR MEMORY CONTENTS
INX	INCREMENT REGISTER PAIR
JC	JUMP IF CARRY
JM	JUMP IF MINUS
JMP	JUMP TO THE INSTRUCTION IDENTIFIED IN THE OPERAND
JNC	JUMP IF NO CARRY
JNZ	JUMP IF NOT ZERO
JP	JUMP IF PLUS
JPE	JUMP IF PARITY EVEN
JPO	JUMP IF PARITY ODD
JZ	JUMP IF ZERO
LDA	LOAD ACCUMULATOR FROM MEMORY USING DIRECT ADDRESSING
LDAX	LOAD ACCUMULATOR FROM MEMORY LOCATION ADDRESSED BY REGISTER PAIR
LHLD	LOAD H AND L REGISTERS DIRECT
LXI	LOAD A 16 BIT VALUE IMMEDIATE INTO A REGISTER PAIR
MOV	MOVE DATA
MVI	LOAD DATA IMMEDIATE INTO REGISTER INTO REGISTER OR MEMORY
NOP	NO OPERATION
ORA	OR REGISTER OR MEMORY WITH ACCUMULATOR
ORI	OR IMMEDIATE WITH ACCUMULATOR
OUT	OUTPUT FROM ACCUMULATOR
PCHL	JUMP TO ADDRESS SPECIFIED BY HL
POP	READ FROM THE TOP OF THE STACK
PUSH	WRITE TO THE TOP OF THE STACK
RAL	ROTATE ACCUMULATOR LEFT THROUGH CARRY
RAR	ROTATE ACCUMULATOR RIGHT THROUGH CARRY
RC	RETURN IF THE CARRY STATUS EQUALS 1
RET	RETURN FROM SUBROUTINE
RIM	READ INTERRUPT MASK
RLC	ROTATE ACCUMULATOR LEFT
RM	RETURN IF THE SIGN STATUS EQUALS 1
RNC	RETURN IF THE CARRY STATUS EQUALS 0
RNZ	RETURN IF THE ZERO STATUS EQUALS 0
RP	RETURN IF THE SIGN STATUS EQUALS 0
RPE	RETURN IF THE PARITY STATUS EQUALS 1

命令	名称
RPO	RETURN IF THE PARITY STATUS EQUALS 0
RRC	ROTATE ACCUMULATOR RIGHT
RST	RESTART
RZ	RETURN IF THE ZERO STATUS EQUALS 1
SBB	SUBTRACT REGISTER OR MEMORY FROM ACCUMULATOR WITH BORROW
SBI	SUBTRACT IMMEDIATE DATA FROM ACCUMULATOR WITH BORROW
SHLD	STORE H AND L REGISTERS DIRECT
SIM	SET INTERRUPT MASK
SPHL	LOAD THE STACK POINTER FROM THE H AND L REGISTERS
STA	STORE ACCUMULATOR IN MEMORY USING DIRECT ADDRESSING
STAX	STORE ACCUMULATOR IN THE MEMORY LOCATION ADDRESSED BY A REGISTER PAIR
STC	SET CARRY STATUS
SUB	SUBTRACT REGISTER OR MEMORY FROM ACCUMULATOR
SUI	SUBTRACT IMMEDIATE DATA FROM ACCUMULATOR
XCHG	EXCHANGE DE AND HL REGISTERS CONTENTS
XRA	EXCLUSIVE OR REGISTER OR MEMORY WITH ACCUMULATOR
XRI	EXCLUSIVE OR IMMEDIATE DATA WITH ACCUMULATOR
XTHL	EXCHANGE TOP OF STACK WITH HL

表で使用されている記号の意味

記号	内容
〈B_2〉	命令の2バイト目の内容
〈B_3〉	命令の3バイト目の内容
r	レジスタA, B, C, D, E, H, Lのうちの1つを示す。
C	キャリーフラグ（オーバーフロウ，アンダーフロウ）
Z	ゼロフラグ（演算結果がゼロ）
S	サインフラグ（最上位ビットが“1”）
P	パリティフラグ（偶数パリティ）
CY_4	ビットナンバー3からの桁上げ
各命令のフラグに対する影響表現方法	
×	フラグが影響をうける
0	フラグをリセットする
1	フラグをセットする
$\bar{C}$	フラグを反転させる
M	レジスタHとLの内容でアドレスされたメモリ番地
(　)	レジスタまたはメモリ番地の内容
AND	論理積
XOR	排他的論理和
OR	論理和
Am	Aレジスタのビットナンバー
SP	スタックポインタ
PC	プログラムカウンタ
←	転　移
SSS	ソースレジスタ
DDD	デスティネーションレジスタ
PSW	アキュムレータとフラグ
〔(　)(　)〕	レジスタ対でアドレスされるメモリ番地の内容
〔〈B_3〉〈B_2〉〕	B_2，B_3の16ビットでアドレスされるメモリ番地の内容
F	フラグの総称
(DBm)	データバスの各ビットの内容
(ABm)	アドレスバスの各ビットの内容

分類	命令の名称	ニーモニック	オペランド形式	インストラクションコード 1バイト目	2バイト目	3バイト目	機能	説明	状態フリップフロップ C	Z	S	P	CY_4
データ転送命令	Move	MOV	r_1, r_2	01DDDSSS			$(r_1) \leftarrow (r_2)$	レジスタr_2の内容をr_1に転送します。r_2の内容はそのままです。					
		MOV	r, M	01DDD110			$(r) \leftarrow [(H)(L)]$	レジスタHとLの内容でアドレスされたメモリの内容をレジスタrに転送します。					
		MOV	M, r	01110SSS			$[(H)(L)] \leftarrow (r)$	レジスタrの内容をレジスタH，Lの内容でアドレスされたメモリに転送します。					
	Move Immediate Data	MVI	r, B_2	00DDD110	B_2		$(r) \leftarrow \langle B_2 \rangle$	$\langle B_2 \rangle$ をレジスタrに転送します。					
		MVI	M, B_2	00110110	B_2		$(M) \leftarrow \langle B_2 \rangle$	$\langle B_2 \rangle$ をレジスタHとLでアドレスされるメモリ番地に転送します。					
	Store Accumulator	STAX	B	00000010			$[(B)(C)] \leftarrow (A)$	レジスタBとCでアドレスされるメモリ番地にレジスタAの内容をストアします。					
		STAX	D	00010010			$[(D)(E)] \leftarrow (A)$	レジスタDとEでアドレスされるメモリ番地にレジスタAの内容をストアします。					
	Load Accumulator	LDAX	B	0001010 0			$(A) \leftarrow [(B)(C)]$	レジスタBとCでアドレスされるメモリ番地の内容をレジスタAに転送します。					
		LDAX	D	00011010			$(A) \leftarrow [(D)(E)]$	レジスタDとEでアドレスされるメモリ番地の内容をレジスタAに転送します。					
	Store Hand L Direct	SHLD	B_3 B_2	00100010	B_2	B_3	$[\langle B_3 \rangle \langle B_2 \rangle] \leftarrow (L)$ $[\langle B_3 \rangle \langle B_2 \rangle + 1] \leftarrow (H)$	レジスタLの内容をメモリ番地B_3，B_2に転送し，レジスタHの内容をメモリ番地B_3B_2+1にストアします。					

分類	命令の名称	ニーモニック	オペランド形式	インストラクションコード 1バイト目	2バイト目	3バイト目	機能	説明	状態フリップフロップ C	Z	S	P	CY_4
データ転送命令	Load H and L Direct	LHLD	B_3 B_2	00101010	B_2	B_3	$(L) \leftarrow [\langle B_3 \rangle \langle B_2 \rangle]$ $(H) \leftarrow [\langle B_3 \rangle \langle B_2 \rangle + 1]$	メモリ番地B_3 B_2 およびB_3 B_2+1の内容をそれぞれレジスタL，Hに転送します。					
	Store Accumulator Direct	STA	B_3 B_2	00110010	B_2	B_3	$[\langle B_3 \rangle \langle B_2 \rangle] \leftarrow (A)$	レジスタAの内容をメモリ番地B_3 B_2にストアします。					
	Load Accumulator H and L	LDA	B_3 B_2	00111010	B_2	B_3	$(A) \leftarrow [\langle B_3 \rangle \langle B_2 \rangle]$	メモリ番地B_3 B_2 の内容をレジスタAにロードします。					
	Load SP from H and L	SPHL		11111001			$(SP) \leftarrow (H)(L)$	レジスタH，Lの内容をスタックポインタに転送します。					
	Load Register Pair Immediate	LXI	$B_1B_3B_2$	00000001	B_2	B_3	$(B) \leftarrow \langle B_3 \rangle$ $(C) \leftarrow \langle B_2 \rangle$	16ビットデータB_3 B_2 をペアレジスタBCに転送します。					
		LXI	$D_1B_3B_2$	00010001	B_2	B_3	$(D) \leftarrow \langle B_3 \rangle$ $(E) \leftarrow \langle B_2 \rangle$	16ビットデータB_3 B_2 をペアレジスタD，Eに転送します。					
		LXI	$H_1B_3B_2$	00100001	B_2	B_3	$(H) \leftarrow \langle B_3 \rangle$ $(L) \leftarrow \langle B_2 \rangle$	16ビットデータB_3 B_2 をペアレジスタH，Lに転送します。					
		LXI	SP，B_3 B_2	00110001	B_2	B_3	$(SP) \leftarrow \langle B_3 \rangle \langle B_2 \rangle$	16ビットデータB_3 B_2 をスタックポインタに転送します。					
	Exchange Registers	XCHG		11101011			$(H) \leftrightarrow (D)$ $(L) \leftrightarrow (E)$	ペアレジスタH，LとペアレジスタDEの内容を交換します。					
	Exchange Stack	XTHL		11100011			$(L) \leftrightarrow [(SP)]$ $(H) \leftrightarrow [(SP)+1]$	ペアレジスタH，Lの内容とスタックポインタでアドレスされるメモリの内容を交換します。					
レジスタ増減命令	Increment Regiser or Memory	INR	r	00DDD100			$(r) \leftarrow (r)+1$	レジスタrの内容を＋1します。		×	×	×	
	Increment Register or Memory	INR	M	00110100			$(M) \leftarrow (M)+1$	レジスタH，Lでアドレスされるメモリ番地の内容を＋1します。		×	×	×	

分類	命令の名称	ニーモニック	オペランド形式	インストラクションコード			機能	説明	状態フリップフロップ				
				1バイト目	2バイト目	3バイト目			C	Z	S	P	CY_4
レジスタ増減命令	Decrement Register or Memory	DCR	r	00DDD101			(r)←(r)−1	レジスタrの内容を−1します。		×	×	×	
		DCR	M	00110101			(M)←(M)−1	レジスタH，Lでアドレスされるメモリ番地の内容を−1します。		×	×	×	
	Increment Register Pair	INX	H	00100011			(H)(L)←(H)(L)+1	ペアレジスタH，Lの内容を+1します。					
		INX	SP	00110011			(SP)←(SP)+1	スタックポインタの内容を+1します。					
		INX	B	00000011			(B)(C)←(B)(C)+1	ペアレジスタB，Cの内容を+1します。					
		INX	D	00010011			(D)(E)←(D)(E)+1	ペアレジスタD，Eの内容を+1します。					
	Decrement Register Pair	DCX	B	00001011			(B)(C)←(B)(C)−1	ペアレジスタB，Cの内容を−1します。					
		DCX	D	00011011			(D)(E)←(D)(E)−1	ペアレジスタD，Eの内容を−1します。					
		DCX	H	00101011			(H)(L)←(H)(L)−1	ペアレジスタH，Lの内容を−1します。					
		DCX	SP	00111011			(D)(E)←(D)(E)−1	スタックポインタの内容を−1します。					
演算命令	Add Register or Memory to Accumulator	ADD	r	10000SSS			(A)←(A)+(r)	レジスタAとrの内容が加算され結果はレジスタAに入ります。	×	×	×	×	×
		ADD	M	10000110			(A)←(A)+(M)	レジスタH，Lでアドレスされるメモリ番地の内容とレジスタAの内容が加算されAに入る。	×	×	×	×	×
	Add Register to Accumulator with Carry	ADC	r	10001SSS			(A)←(A)+(r)+C	レジスタAとrの内容およびキャリの加算が行われ結果はレジスタAに入ります。	×	×	×	×	×

分類	命令の名称	ニーモニック	オペランド形式	インストラクションコード			機能	説明	状態フリップフロップ				
				1バイト目	2バイト目	3バイト目			C	Z	S	P	CY4
演算・論理操作命令	Add Memory to Accumulator with Carry	ADC	M	10001110			(A)←(A)+(M)+C	レジスタH，Lでアドレスされるメモリの内容とレジスタAとキャリの加算の結果がレジスタAに入ります。	×	×	×	×	×
	Subtract Register or Memory from Accumulator	SUB	r	10010SSS			(A)←(A)−(r)	レジスタAの内容からレジスタrの内容が引かれ結果はレジスタAに入ります。	×	×	×	×	×
		SUB	M	10010110			(A)←(A)−(M)	レジスタAからレジスタH，Lでアドレスされるメモリの内容が引かれ結果はレジスタに入ります。	×	×	×	×	×
	Subtract Register or Memory from Accumulator with Borrow	SBB	r	10011SSS			(A)←(A)−(r)−C	レジスタrとキャリの加算された値がレジスタAから引かれ結果はレジスタAに入ります。	×	×	×	×	×
		SBB	M	10011110			(A)←(A)−(M)−C	レジスタH，Lでアドレスされるメモリの内容とキャリの加算された値がレジスタAから引かれ結果はレジスタAに入ります。	×	×	×	×	×
	Logical AND Register or Memory with Accumulator	ANA	r	10100SSS			(A)←(A)AND(r) C←"0"	レジスタAとrの各ビット間の論理積をとり結果はレジスタAに入ります。キャリはリセットされます。	0	×	×	×	
		ANA	M	10100110			(A)←(A)AND(M) C←"0"	レジスタAとレジスタH，Lでアドレスされるメモリの内容の論理積をとり結果はレジスタAに入ります。キャリはリセットされます。	0	×	×	×	
	Logical Exclusive OR Register or Memory with Accumulator (Zero Acc)	XRA	r	10101SSS			(A)←(A)XOR(r) C←"0"	レジスタAとrの排他論理和がとれ結果はレジスタAに入ります。キャリはリセットされます。	0	×	×	×	
		XRA	M	10101110			(A)←(A)XOR(M) C←"0"	レジスタAとレジスタH，Lでアドレスされるメモリの内容の排他論理和をとり，結果はレジスタAに入ります。キャリはリセットされる。	0	×	×	×	

分類	命令の名称	ニーモニック	オペランド形式	インストラクションコード			機能	説明	状態フリップフロップ				
				1バイト目	2バイト目	3バイト目			C	Z	S	P	CY_4
演算・論理操作命令	Logical OR Register or Memory with Accumulator	ORA	r	10110SSS			(A)←(A)OR(r) C←"0"	レジスタAとrの論理和をとり，結果はレジスタAに入ります。 キャリはリセットされます。	0	×	×	×	
		ORA	M	10110110			(A)←(A)OR(M) C←"0"	レジスタAとレジスタH，Lでアドレスされるメモリの内容の論理和をとり結果はレジスタAに入ります。キャリはリセットされる。	0	×	×	×	
	Compare Register or Memory with Accumulator	CMP	r	10111SSS			(A)−(r)	レジスタAとrの内容を比較します。両方とも内容は変化しません。減算の結果の判定はフラグで行ないます。	×	×	×	×	×
		CMP	M	10111110			(A)−(M)	レジスタAとレジスタH，Lでアドレスされるメモリの内容の比較をします。両方とも内容は変化しません。	×	×	×	×	×
	Add Immediate to Accumulator	ADI	B_2	11000110	B_2		(A)←(A)+〈B_2〉	レジスタAの内容とデータB_2の加算後結果はレジスタAに入ります。	×	×	×	×	×
	Add Immediate to Accumulator with Carry	ACI	B_2	11000110	B_2		(A)←(A)+〈B_2〉+C	レジスタAとデータB_2とキャリの加算後結果はレジスタAに入ります。	×	×	×	×	×
	Subtract Immediate from Accumulator	SUI	B_2	11010110	B_2		(A)←(A)−〈B_2〉	レジスタAからデータB_2を減算して結果をレジスタAに入れます。	×	×	×	×	×
	Subtract Immediate from ACC with Borrow	SBI	B_2	11011110	B_2		(A)←(A)−〈B_2〉−C	データB_2とキャリが加算された値をレジスタAから減算します。	×	×	×	×	×
	AND Immediate with Accumulator	ANI	B_2	11100110	B_2		(A)←(A)AND〈B_2〉 C←"0"	レジスタAとデータB_2の各ビット間の論理積をとり，レジスタAに入れます。キャリはリセットされます。	0	×	×	×	
	Exclusive −OR Immediate with Accumulator	XRI	B_2	11101110	B_2		(A)←(A)XOR〈B_2〉 C←"0"	レジスタAとデータB_2の各ビット間の排他論理和をとり，レジスタAに入れます。キャリはリセットされます。	0	×	×	×	

分類	命令の名称	ニーモニック	オペランド形式	インストラクションコード			機能	説明	状態フリップフロップ				
				1バイト目	2バイト目	3バイト目			C	Z	S	P	CY4
演算・論理操作命令	OR Immediate with Accumulator	ORI	B_2	11110110	B_2		$(A) \leftarrow (A) OR \langle B_2 \rangle$ $C \leftarrow$ "0"	レジスタAとデータB_2の論理和がとられ結果はレジスタAに入ります。キャリフラグはリセットされます。	0	×	×	×	
	Compare Immediate with Accumulator	CPI	B_2	11111110	B_2		$(A) - \langle B_2 \rangle$	レジスタAとデータB_2の比較をします。減算後レジスタAの内容は変化しません。結果の判定はフラグで行ないます。	×	×	×	×	×
	Double Add	DAD	B	00001001			$(H)(L) \leftarrow (H)(L) + (B)(C)$	ペアレジスタH，LとB，Cの加算後結果はペアレジスタH，Lに入ります。	×				×
		DAD	D	00011001			$(H)(L) \leftarrow (H)(L) + (D)(E)$	ペアレジスタH，LとD，Eの加算後結果はペアレジスタH，Lに入ります。	×				×
		DAD	H	00101001			$(H)(L) \leftarrow (H)(L) + (H)(L)$	ペアレジスタH，Lの内容が2倍されます。	×				×
		DAD	SP	00111001			$(H)(L) \leftarrow (H)(L) + (SP)$	ペアレジスタH，Lの内容とスタックポインタの内容が加算され，H，Lに結果が入ります。	×				×
回転命令	Rotate Accumulator Left	RLC		00000111			$(Am_{+1}) \leftarrow (Am)$ $(A_0) \leftarrow (A_7), C \leftarrow (A_7)$	レジスタAの内容は1ビット左シフトされL．S．BとキャリにはM．S．Bの値が入ります。	×				
	Rotate Accumulator Right	RRC		00001111			$(Am_{-1}) \leftarrow (Am)$ $(A_7) \leftarrow (A_0), C \leftarrow (A_0)$	レジスタAの内容は1ビット右シフトされM．S．BとキャリにはL．S．Bの値が入ります。	×				
	Rotate Accumulator Left through Carry	RAL		00010111			$(Am_{+1}) \leftarrow (Am)$ $(A_0) \leftarrow C, C \leftarrow (A_7)$	レジスタAの内容がキャリを含めて1ビット左回転されます。	×				
	Roatte Accumulator Right through Carry	RAR		00011111			$(Am_{-1}) \leftarrow (Am)$ $(A_7) \leftarrow C, C \leftarrow (A_0)$	レジスタAの内容がキャリを含めて1ビット右回転されます。	×				

分類	命令の名称	ニーモニック	オペランド形式	インストラクションコード			機能	説明	状態フリップフロップ				
				1バイト目	2バイト目	3バイト目			C	Z	S	P	CY_4
アキュムレータ補正命令	Decimal Adjust Accumulator	DAA		00100111			10進加減算補正	レジスタAの8ビットを2桁のBCDコードに変換します。	×	×	×	×	
	Complement Accumulator	CMA		00101111			$(A) \leftarrow (\overline{A})$	レジスタAの内容の各ビットを反転させます。					
キャリ操作命令	Set Carry	STC		00110111			C←"1"	キャリフラグがセットされます。	1				
	Complement Carry	CMC		00111111			$C \leftarrow \overline{C}$	キャリフラグを反転させます。	×				
割込制御命令	Disable Interrupt	DI		11110011				内部割込許可フリップフロップをリセットして割込を禁止します。					
	Enable Interrupt	EI		11111011				内部割込許可フリップフロップをセットして割込可能な状態にします。					
分岐命令	Jump	JMP	$B_3 B_2$	11000011	B_2	B_3	$(PC) \leftarrow \langle B_3 \rangle \langle B_2 \rangle$	プログラムカウンタに16ビットアドレスB_3 B_2をセットしてB_3 B_2でアドレスされるメモリ番地の命令に無条件でジャンプします。					
	Jump if Not Zero	JNZ	$B_3 B_2$	11000010	B_2	B_3	Z="0", $(PC) \leftarrow \langle B_3 \rangle \langle B_2 \rangle$ Z="1", $(PC) \leftarrow (PC)+3$	Zフラグが"0"ならば上位アドレスB_3，下位アドレスB_2でアドレスされるメモリ番地の命令にジャンプします。					
	Jump if Zero	JZ	$B_3 B_2$	11001010	B_2	B_3	Z="1", $(PC) \leftarrow \langle B_3 \rangle \langle B_2 \rangle$ Z="0", $(PC) \leftarrow (PC)+3$	Zフラグが"1"ならば上位アドレスB_3，下位アドレスB_2でアドレスされるメモリ番地の命令にジャンプします。					
	Jump if No Carry	JNC	$B_3 B_2$	11010010	B_2	B_3	C="0", $(PC) \leftarrow \langle B_3 \rangle \langle B_2 \rangle$ C="1", $(PC) \leftarrow (PC)+3$	Cフラグが"0"ならば上位アドレスB_3，下位アドレスB_2でアドレスされるメモリ番地の命令にジャンプします。					

分類	命令の名称	ニーモニック	オペランド形式	インストラクションコード			機能	説明	状態フリップフロップ				
				1バイト目	2バイト目	3バイト目			C	Z	S	P	CY_4
	Jump if Carry	JC	B_3 B_2	11011010	B_2	B_3	C＝“1”, (PC)←⟨B_3⟩⟨B_2⟩ C＝“0”, (PC)←(PC)＋B	Cフラグが“1”ならば上位アドレスB_3，下位アドレスB_2でアドレスされるメモリ番地の命令にジャンプします。					
分	Jump if Parity Odd	JPO	B_3 B_2	11100010	B_2	B_3	P＝“0”, (PC)←⟨B_3⟩⟨B_2⟩ P＝“1”, (PC)←(PC)＋3	Pフラグが“0”ならば上位アドレスB_3，下位アドレスB_2 でアドレスされるメモリ番地の命令にジャンプします。					
岐	Jump if Parity Even	JPE	B_3 B_2	11101010	B_2	B_3	P＝““1”, (PC)←⟨B_3⟩⟨B_2⟩ P＝“0”, (PC)←(PC)＋3	Pフラグが“1”ならば上位アドレスB_3，下位アドレスB_2 でアドレスされるメモリ番地の命令にジャンプします。					
命	Jump if Positive	JP	B_3 B_2	11110010	B_2	B_3	S＝“0”, (PC)←⟨B_3⟩⟨B_2⟩ S＝“1”, (PC)←(PC)＋3	Sフラグの内容が“0”ならば上位アドレスB_3、下位アドレスB_2でアドレスされるメモリ番地の命令にジャンプします。					
令	Jump if Minus	JM	B_3 B_2	11111010	B_2	B_3	S＝“1”, (PC)←⟨B_3⟩⟨B_2⟩ S＝“0”, (PC)←(PC)＋3	Sフラグの内容が1ならば上位アドレスB_3，下位アドレスB_2 でアドレスされるメモリ番地の命令にジャンプします。					
	Load Program Counter	PCHL	B_3 B_2	11101001	B_2	B_3	(PC)←(H)(L)	ペアレジスタH，Lの内容をプログラムカウンタに転送して，HLでアドレスされるメモリ番地のインストラクションにジャンプします。					
サブルーチンコール命令	Call	CALL	B_3 B_2	11001101	B_2	B_3	〔(SP)−1〕〔(SP)−2〕 ←(PC) (SP)←(SP)−2 (PC)←⟨B_3⟩⟨B_2⟩	スタック・ポインタでアドレスされるスタックにプログラム・カウンタの内容をストアしてB_3 B_2 で示されるアドレスにあるサブルーチンにジャンプします。					

分類	命令の名称	ニーモニック	オペランド形式	インストラクションコード			機能	説明	状態フリップフロップ				
				1バイト目	2バイト目	3バイト目			C	Z	S	P	CY_4
サブルーチンコール命令	Call if Not Zero	CNZ	B_3 B_2	11000100	B_2	B_3	Z＝“0”, [(SP)−1][(SP)−2] ←(PC) (SP)←(SP)−2 (PC)←⟨B_3⟩⟨B_2⟩ Z＝“1” (PC)←(PC)+3	Zフラグの内容が“0”ならばプログラム・カウンタの内容をスタックして上位アドレスB_3，下位アドレスB_2に示されるアドレスのサブルーチンにジャンプします。					
	Call if Zero	CZ	B_3 B_2	11001100	B_2	B_3	Z＝“1”, [(SP)−1][(SP)−2] ←(PC) (SP)←(SP)−2 (PC)←⟨B_3⟩⟨B_2⟩ Z＝“0”, (PC)←(PC)+3	Zフラグの内容が“1”ならばプログラム・カウンタの内容をスタックして上位アドレスB_3，下位アドレスB_2で示されるアドレスのサブルーチンにジャンプします。					
	Call if No Carry	CNC	B_3 B_2	11010100	B_2	B_3	C＝“0”, [(SP)−1][(SP)−2] ←(PC) (SP)←(SP)−2 (PC)←⟨B_3⟩⟨B_2⟩ C＝“1”, (PC)←(PC)+3	Cフラグの内容が“0”ならばプログラム・カウンタの内容をスタックして上位アドレスB_3，下位アドレスB_2で示されるアドレスのサブルーチンにジャンプします。					
	Call if Carry	CC	B_3 B_2	11011100	B_2	B_3	C＝“1”, [(SP)−1][(SP)−2] ←(PC) (SP)←(SP)−2 (PC)←⟨B_3⟩⟨B_2⟩ C＝“0”, (PC)←(PC)+3	Cフラグの内容が“1”ならばプログラム・カウンタの内容をスタックして上位アドレスB_3，下位アドレスB_2で示されるアドレスのサブルーチンにジャンプします。					
	Call if Parity Odd	CPO	B_3 B_2	11100100	B_2	B_3	P＝“0”, [(SP)−1][(SP)−2] ←(PC) (SP)←(SP)−2 (PC)←⟨B_3⟩⟨B_2⟩ P＝“1”, (PC)←(PC)+3	Pフラグの内容が“0”ならばプログラム・カウンタの内容をスタックして上位アドレスB_3，下位アドレスB_2で示されるアドレスのサブルーチンにジャンプします。					

分類	命令の名称	ニーモニック	オペランド形式	インストラクションコード			機能	説明	状態フリップフロップ				
				1バイト目	2バイト目	3バイト目			C	Z	S	P	CY_4
サブルーチンコール命令	Call if Parity Even	CPE	B_3 B_2	11101100	B_2	B_3	P＝“1”, 〔(SP)−1〕〔(SP)−2〕 ←(PC) (SP)←(SP)−2 (SP)←⟨B_3⟩⟨B_2⟩ P＝“0”, (PC)←(PC)＋3	Pフラグの内容が“1”ならばプログラム・カウンタの内容をスタックして上位アドレスB_3，下位アドレスB_2で示されるアドレスのサブルーチンにジャンプします。					
	Call if Positive	CP	B_3 B_2	11110100	B_2	B_3	S＝“0”, 〔SP−1〕〔SP−2〕 ←(PC) (SP)←(SP)−2 (PC)←⟨B_3⟩⟨B_2⟩ S＝“1”, (PC)←(PC)＋3	Sフラグの内容が“0”ならばプログラム・カウンタの内容をスタックして上位アドレスB_3，下位アドレスB_2で示されるアドレスのサブルーチンにジャンプします。					
	Call if Minus	CM	B_3 B_2	11111100	B_2	B_3	S＝“1”, 〔SP−1〕〔SP−2〕 ←(PC) (SP)←(SP)−2 (PC)←⟨B_3⟩⟨B_2⟩ S＝“0”, (PC)←(PC)＋3	Sフラグの内容が“1”ならばプログラム・カウンタの内容をスタックして上位アドレスB_3，下位アドレスB_2で示されるアドレスのサブルーチンにジャンプします。					
リターン命令	Return	RET		11001001			(PC)←〔SP+1〕〔SP+2〕 (SP)←(SP)＋2	スタックポインタでアドレスされるプッシュダウンスタックにストアされている値によってアドレスされるメモリ番地の命令に戻ります。					
	Return if Not Zero	RNZ		11000000			Z＝“0”, (PC)←〔SP+1〕 〔SP+2〕 (SP)←(SP)＋2 Z＝“1” (PC)←(PC)＋1	Zフラグの内容が“1”ならばスタックしておいたアドレスで示されるメモリ番地の命令に戻ります。					

分類	命令の名称	ニーモニック	オペランド形式	インストラクションコード			機能	説明	状態フリップフロップ				
				1バイト目	2バイト目	3バイト目			C	Z	S	P	CY_4
リターン命令	Return if Zero	RZ		11001000			Z＝“1”, (PC)←〔SP+1〕〔SP+2〕 (SP)←(SP)+2 Z＝“0”, (PC)←(PC)+1	Zフラグの内容が“1”ならばスタックしておいたアドレスで示されるメモリ番地の命令に戻ります。					
	Return if No Carry	RNC		11010000			C＝“0”, (PC)←〔SP+1〕〔SP+2〕 (SP)←(SP)+2 C＝“1”, (PC)←(PC)+1	Cフラグの内容が“0”ならばスタックしておいたアドレスで示されるメモリ番地の命令に戻ります。					
	Return if Carry	RC		11011000			C＝“1”, (PC)←〔SP+1〕〔SP+2〕 (SP)←(SP)+2 C＝“0”, (PC)←(PC)+1	Cフラグの内容が“1”ならばスタックしておいたアドレスで示されるメモリ番地の命令に戻ります。					
	Return if Parity Odd	RPO		11100000			P＝“0”, (PC)←〔SP+1〕〔SP+2〕 (SP)←(SP)+2 P＝“1”, (PC)←(PC)+1	Pフラグの内容が“0”ならばスタックしておいたアドレスで示されるメモリ番地の命令に戻ります。					
	Return if Parity Even	RPE		11101000			P＝“1”, (PC)←〔SP+1〕〔SP+2〕 (SP)←(SP)+2 P＝“0”, (PC)←(PC)+1	Pフラグの内容が“1”ならばスタックしておいたアドレスで示されるメモリ番地の命令に戻ります。					

分類	命令の名称	ニーモニック	オペランド形式	インストラクションコード			機能	説明	状態フリップフロップ				
				1バイト目	2バイト目	3バイト目			C	Z	S	P	CY
リターン命令	Return if Positive	RP		11110000			S＝“0”, (PC)←〔(SP)+1〕〔(SP)+2〕 (SP)←(SP)+2 S＝“1”, (PC)←(PC)+1	Sフラグの内容が“0”ならばスタックしておいてアドレスで示されるメモリ番地の命令に戻ります。					
	Return if Minus	RM		11111000			S＝“1”, (PC)←〔(SP)+1〕〔(SP)+2〕 (SP)←(SP)+2 S＝“0” (PC)←(PC)+1	Sフラグの内容が“1”ならばスタックしておいたアドレスで示されるメモリ番地の命令に戻ります。					
スタック操作命令	Push Data onto Stack	PUSH	B	11000101			〔(SP)−1〕←(B) 〔(SP)−2〕←(C) (SP)←(SP)−2	ペアレジスタB，Cの内容がスタックポインタでアドレスされるプッシュダウン・スタックにストアされます。					
		PUSH	D	11010101			〔(SP)←1〕←(D) 〔(SP)−2〕←(E) (SP)←(SP)−2	ペアレジスタD，Eの内容がスタックポインタでアドレスされるプッシュダウン・スタックにストアされます。					
		PUSH	H	11100101			〔(SP)−1〕←(H) 〔(SP)−2〕←(L) (SP)←(SP)−2	ペアレジスタH，Lの内容がスタックポインタでアドレスされるプッシュダウン・スタックにストアされます。					
		PUSH	PSW	11110101			〔(SP)−1〕←(A) 〔(SP)−2〕←(F) (SP)←(SP)−2	レジスタAの内容とフラグの内容がスタック・ポインタでアドレスされるプッシュダウン・スタックにストアされます。					
	Pop Data Off Stack	POP	B	11000001			(C)←〔(SP)〕 (B)←〔(SP)+1〕 (SP)←(SP)+2	スタック・ポインタでアドレスされるプッシュダウン・スタックの内容をペアレジスタB，Cにロードします。					
		POP	D	11010001			(E)←〔(SP)〕 (D)←〔(SP)+1〕 (SP)←(SP)+2	スタック・ポインタでアドレスされるプッシュダウン・スタックの内容をペアレジスタD，Eにロードします。					

分類	命令の名称	ニーモニック	オペランド形式	インストラクションコード			機能	説明	状態フリップフロップ				
				1バイト目	2バイト目	3バイト目			C	Z	S	P	CY_4
スタック操作命令	Pop Data Off Stack	POP	H	11100001			(L)←[(SP)] (H)←[(SP)+1] (SP)←(SP)+2	スタック・ポインタでアドレスされるプッシュダウン・スタックの内容をペアレジスタH，Lにロードします。					
		POP	PSW	11110001			(F)←[(SP)] (A)←[(SP)+1] (SP)←(SP)+2	スタック・ポインタでアドレスされるプッシュダウン・スタックの内容をフラグおよびレジスタAにそれぞれロードします。	×	×	×	×	×
入出力命令	Output	OUT	B_2	11010011	B_2		$(DB_{0\sim7})$←(A) $(AB_{8\sim15})$←(F) $(AB_{0\sim7})$←$\langle B_2\rangle$ 注	レジスタAの内容をデータバスに出力すると共にアドレスバスの上位8ビットにはフラグの内容が下位8ビットにはデバイス番号を示す$\langle B_2\rangle$が出されます。 注					
	Input	IN	B_2	11011011	B_2		$(AB_{8\sim15})$←(A) $(AB_{0\sim7})$←$\langle B_2\rangle$ (A)←$(DB_{0\sim7})$ 注	アドレスバスの上位8ビットにレジスタAの内容を，下位8ビットにはデバイス番号を示す$\langle B_2\rangle$が出され，データバスの内容をレジスタAに読込みます。 注					
リスタート命令	Restart	RST	X	11AAA111 (AAA=X)			[(SP)−1] [(SP)−2]←(PC) (SP)←(SP)−2 (PC)←⟨000000000 0AAA000⟩	プログラムカウンタの内容をスタックポインタで示されるプッシュダウン・スタックにストアしてプログラムカウンタには8進の0000X0をロードしてその番号からプログラムをスタートさせる。					
その他の命令	Halt	HLT		01110110			HALT	CPUはHALT状態になります。データバス，アドレスバスはフローティング状態になります。					
	No Operation	NOP		00000000			NO OPERATION	CPUは何の動作もせず，ただ1マシンサイクルの時間を消費します。					

注．上述の説明はD753用でD8080Aでは，IN／OUT命令とも$AB_{8\sim15}$にも$\langle B_2\rangle$が出力されます。

コントロール・コードの意味とキーとの対応表

シンボル（PC内シンボル）	16 進	Control をおしながら	機　能	意　味（PC内の意味）
NUL	0 0	スペース	null	空白
SOH (SH)	0 1	A	start of heading	ヘッディング開始
STX (SX)	0 2	B	start of text	テキスト開始(カーソル左移動)
ETX (EX)	0 3	C	end of text	テキスト終結(実行中止)
EOT (ET)	0 4	D	end of transmission	伝送終了
ENQ (EQ)	0 5	E	enquiry	問合せ(カーソル位置抹消)
ACK (AK)	0 6	F	acknowledge	肯定応答
BEL (BL)	0 7	G	bell	ベル
BS (BS)	0 8	H	back space	後退(1文字削除)
HT (HT)	0 9	I	horizontal tabulation	水平タブ
LF (LF)	0 A	J	line feed	改行
VT (HM)	0 B	K	vertical tabulation	垂直タブ(ホームポジション)
FF (CL)	0 C	L	form feed	書式送り(画面クリア)
CR (CR)	0 D	M	carriage return	復帰
SO (SO)	0 E	N	shift out	シフトアウト(カーソル右移動)
SI (SI)	0 F	O	shift in	シフトイン
DLE (DE)	1 0	P	data link escape	伝送制御拡張
DC1 (D1)	1 1	Q	device control 1	装置制御1
DC2 (D2)	1 2	R	device control 2	装置制御2(1文字挿入)
DC3 (D3)	1 3	S	device control 3	装置制御3
DC4 (D4)	1 4	T	device control 4	装置制御4
NAK (NK)	1 5	U	negative acknowledge	否定応答
SYN (SN)	1 6	V	synchronous idle	同期信号
ETB (EB)	1 7	W	end of transmission block	伝送ブロック終結
CAN (CN)	1 8	X	cancel	取消し
EM (EM)	1 9	Y	end of mediunm	媒体終端
SUB (SB)	1 A	Z	substitute	文字置換
ESC (EC)	1 B		escape	拡張
FS (→)	1 C		file separator	ファイル分離
GS (←)	1 D		group separator	グループ分離
RS (↑)	1 E		record separator	レコード分離
US (↓)	1 F		unit separator	ユニット分離

ＰＣ－8801　キャラクタ・コード表

上位４ビット→（列）／下位４ビット↓（行）

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		D_E		0	@	P		p				ー	タ	ミ		
1	S_H	D_1	!	1	A	Q	a	q			。	ア	チ	ム		円
2	S_X	D_2	"	2	B	R	b	r			「	イ	ツ	メ		年
3	E_X	D_3	#	3	C	S	c	s			」	ウ	テ	モ		月
4	E_T	D_4	$	4	D	T	d	t			、	エ	ト	ヤ		日
5	E_Q	N_K	%	5	E	U	e	u			・	オ	ナ	ユ		時
6	A_K	S_N	&	6	F	V	f	v			ヲ	カ	ニ	ヨ		分
7	B_L	E_B	▼	7	G	W	g	w			ァ	キ	ヌ	ラ		秒
8	B_S	C_N	(	8	H	X	h	x			ィ	ク	ネ	リ	♠	
9	H_T	E_M	)	9	I	Y	i	y			ゥ	ケ	ノ	ル	♥	
A	L_F	S_B	*	:	J	Z	j	z			ェ	コ	ハ	レ	♦	
B	H_M	E_C	+	;	K	[	k	{			ォ	サ	ヒ	ロ	♣	
C	C_L	→	,	<	L	¥	l	|			ャ	シ	フ	ワ	●	
D	C_R	←	−	=	M	]	m	}			ュ	ス	ヘ	ン	○	
E	S_O	↑	.	>	N	∧	n	~			ョ	セ	ホ	゛		
F	S_I	↓	／	?	O	—	o				ッ	ソ	マ	゜		

著者略歴

脇　英世（わき　ひでよ）
1947年東京生まれ。早稲田大学理工学部電子通信学科卒業。同大学大学院博士課程修了。工学博士。
現在，東京電機大学工学部電気通信工学科助教授。
著書に
『マイコン標準インターフェース』オーム社。『ビジネスマンのための実践ベーシック教室』東洋経済新報社，『ＦＭ８操縦法』ラジオ技術社，『マイコンによる知的生産の技術』講談社，『日本語ワード・プロセッサー入門』講談社。

PC-8801
アセンブリ言語プログラミング入門

著　者　脇　英世

発行者　田村正隆

発行所　株式会社　ナツメ社
東京都千代田区神田神保町1　―52（〒101）
電話　03（291）1257（代表）
振替　東京３―58661
印　刷　ラン印刷社
製　本　文章堂製本

ＩＳＢＮ4-8163-0283-2　Ｃ2054　Printed in Japan

PC-8801
プログラミング入門

PC-8001の上位機種として脚光をあびているPC-8801に内容を限定した。
はじめてパソコンにさわるという人を対象に、前半では電源の入れ方、キー操作の方法といった初歩的な知識をもてるようにした。後半はこの機種がビジネス志向であることをふまえ、経営分析、設備投資計画など、経営的なプログラムに多くのページを割いた。

好評発売中

野々山隆幸著
菊判　200頁　1400円

PC-8801
ビジネス・プログラミング

パソコンをビジネスに利用したいという方のために、いかにパソコンを利用するかという問題の提起・把握から、それを解決するためのシステムの設計と、それにもとづくプログラムの作成まで、具体例をあげて解説した。この本でビジネス・プログラミング自在。

布田　治著
菊判　216頁　1500円

PC-8801
アセンブリ言語プログラミング入門

発行――――昭和58年 3 月10日
著者――――脇英世
発行者―――田村正隆
発行所―――株式会社ナツメ社
郵便番号101
東京都千代田区神田神保町1-52
電話<03>291-1257
振替　東京3-58661
<落丁・乱丁本はお取り替えします>

定価――――**1600円**

PC-8801

アセンブリ言語プログラミング入門

［目次］より

ナツメ社
定価＝1600円

Personal Computer

PC-8801

NEC

アセンブリ言語プログラミング入門

脇英世＝著

ナツメ社

ISBN4-8163-0283-2 C2054 ¥1600E